Data Management and Concurrency Control in Broadcast based Asymmetric Environments

Thesis in partial fulfilment of the degree of

Master in Technology in

Information and Communication Technology

Spring 2006

Agder University College

Faculty of Engineering and Science

Huazhong University of Science and Technology

School of Computer Science and Technology

Supervisors:

Professor Guohui Li (HUST)

Ass. Prof. Ole-Christoffer Granmo (HiA)

Authors:

Arild Finne

Erik Trædal

Pages: 70 (including this page)

Modified date: 2006-09-18

Keywords: Concurrency Control, Data Management, Mobile Computing, Data Dissemination, Data Broadcast

Abstract:

Tens of millions of users have personal handheld devices with several network interfaces built-in, and the number of users and of network interfaces included are only increasing. This growth suggests a need for new methods to disseminate data to multiple clients, and cyclic broadcast is one approach. We do a survey on the various data management protocols that describe how to broadcast the data, and the concurrency control protocols that make sure all access to the database is consistent. The various data management and concurrency control techniques deals with the restrictions in asymmetric broadcast environments and improves the performance. As part of the survey, we made a simulation platform and implemented several data management techniques and four concurrency protocols, BCC-TI, FBOCC, PVTO and STUBcast. We also implemented a technique we call "partial restart" *into FBOCC. Simulation results shows nearly a 30% decrease in transaction execution time when the transaction length is 6. The survey and the simulations helped us to make a qualitative characterization framework, which categorize the protocols based on the environments they apply to and qualities they possess. We offer the source code to the simulation models and the protocols we implemented which can be used as a base to implement new protocols.*

Project home page: http://www.arild.finne.googlepages.com/masterthesis.html

Table of Contents

List of Tables

List of Illustrations

1 Executive Summary

New to this topic, we first had to study a big amount of literature, then we wrote a survey as an introduction for others, made a simulation model and implemented some state-of-the-art protocols and analysed the simulation results. We present a framework in order to categorize the protocols based on their qualities, and we suggest one improvement to the existing protocols.

The environment of interest has asymmetric bandwidth, which means the bandwidth from the client to the server (upload speed) is much smaller then the bandwidth from the server to the client (download speed). When the server (information source) broadcasts a message, all the clients will be able to receive this one message. The typical example of this kind of environment are wireless networks. And because most wireless networks have mobile handheld units as clients, restrictions such as low processing ability and limited battery power must be considered.

The data management defines how the data should be broadcasted. Three fundamental ways to do it is push based, pull based or a hybrid of the two first. Pull based data dissemination is broadcasting of data which are specifically requested, where as push based is to broadcast all information in cycles. The latter has received most attention because when most clients only read data (which is in the definition of data dissemination), the push approach is very scalable with respect to the number of clients.

To prevent the clients from wasting battery power when listening to the broadcast channel in search of a data value, an *index* is broadcasted for each cycle. After it has read the *index* and found the time when the desired value is broadcasted, the clients can turn off the wireless interface while waiting. The time it takes for a client to find out when a data item is broadcasted (the time it takes to find the index and read the index), is called *tuning time*. The time it takes to find, wait for, and read a data value is called *latency time*. Other data management techniques are caching, compression, prefetching and data dissemination.

Broadcast disks is an approach that assume some data is more frequently accessed then others. The data is divided into two or more disks based on their probability of being used, and the frequently accessed disk is broadcasted more often then the low probability accessed data.

The concurrency control protocol makes sure the clients get a consistent view of the data set, when the data base set is updated, either by the server itself or other clients in between the operations. The level of consistency is defined by a correctness criteria. The normal notion is serializability, but to achieve better performance some protocols choose a weakened correctness criteria. Also, some protocols support real time, which mean transactions can be prioritized based on time restrictions and deadlines.

We made a simulation platform in order to test and compare four protocols, BCC-TI, FBOCC, PVTO and STUBcast. The protocols represent a wide spectre regarding the various environments they apply to. We discovered that FBOCC performs very good in most environments. BCC-TI performs best in client read-only environments, because BCC-TI is made for read-only. STUBcast uses a relaxed correctness criteria (which makes it accept more transaction) so it has the best performance, especially on long transactions. But STUBcast has several drawbacks that makes it not suitable for battery powered devices. Our final discovery was the usage of a new approach we name *partial restart*. Partial restart will only rollback to the operation that causes the conflict instead of restarting all the operations in a transaction. The simulations result showed a good improvement, especially for long transactions (almost 50% shorter transaction execution time when the transaction length is 12).

Based on the test results and the survey, we made a qualitative characterization framework. It categorize the protocols based on their qualities. For data management protocols the following characterization criteria were chosen, *Method*, *Latency*, *Processing*, *Size Increase* and *Tuning Time*. *Processing* defines the amount of CPU power the protocols need and *Tuning time* defines how long a client has to listen on the broadcast to find an item. *Electrical power usage*, *Performance*, *Real-time*, *Correctness Criteria* and *Client Update Transactions* were chosen for the concurrency protocols. *Electrical power usage* indicates how battery consuming the protocol is (by looking at network and CPU usage) and *Performance* only indicates the performance relatively to the other protocols.

We have succeeded in getting familiar with the current topic, and even managed to find a new approach to improve performance. The survey gives a soft introduction at the same time as it goes deeper into some parts. The focus is on the performance increasing techniques and categorizing so it is easier to get an overview. The framework follow up this categorizing. The simulation platform will be useful in future research, and we regret we did not have more time to test out and compare even more protocols. But extensive simulation testing was out of the scope.

2 Introduction

In the upcoming sections we give a little introduction to the research area of broadcast based asymmetric environments. In chapter 2.1 we discuss the motivation for doing research on this topic. Chapter 2.2 gives a quick review of earlier published research on the topic. In chapter 2.3 we present our claim and the project definition. Defining what this project try to accomplish. And chapter 2.4 gives explains how this report is structured.

2.1 Motivation

Tens of millions of users have personal handheld devices with several network interfaces built-in, and the number of users and of network interfaces included are only increasing. At the same time the availability of wireless networks increases, so the users use them more frequently. This leads to a higher load on the servers providing the information, and the networks transferring it. Especially in high density populated areas, the number of clients (users) can be very high, so an efficient technique to disseminate data should be found.

The problem of many users is not restricted to wireless networks but is also present in wired networks, where both multiple servers and big bandwidth lines are needed in order to serve all the clients. The problem is just more visible in wireless networks because the bandwidth is smaller and physically limited in respect of available frequencies. Wired networks have in general a lot bigger bandwidth, and can add another physical link if more bandwidth is needed. Wireless networks have additional limitations due to the mobile client devices such as limited battery and processing power.

Data dissemination is defined as the delivery of data from a set of producers to a larger set of clients where the bandwidth is asymmetric. All data disseminating applications communicates in a pattern with a small data packet requesting for information followed by a big sized reply containing the information. An example is a traffic information system. A user requests traffic information for a specific area, and receives data describing the traffic in the whole area.

The solution in handling the big group of clients, without congesting the network or overloading the server (information provider), is to broadcast all the information continuously in cycles. This push based data dissemination is particularly scalable when the clients mostly access the data in a read-only fashion.

The wireless networks have physical support for broadcast and the data dissemination needed by most applications mainly consists of read-only accesses, so the solution fits good to the wireless environment.

2.2 Review

Data dissemination through cyclic broadcasting is an old technology, used for instance in Teletext (reviewed in [1] and chapter 3.7), invented and launched in the 70's [2]. In 1987, Herman et al. [3] introduced *Datacycle* for broadcasting in high throughput wired networks. But it was first in 1994 when Imielinski et. al. [4] introduced a cyclic way to broadcast data in wireless environments, that the research in this area was intensified. Soon after, Acharya et. al. [5] extended the work of Imielinski et al. and named it broadcast disks. They introduced caching and multiple disks to take into account none uniform access patterns. While several different ways to improve indexing and caching has been proposed, broadcast disks has been a kind of standard in this research area. But in 2005 Chang et. all. [6] improved broadcast disks by solving an empty slot problem that could occur under certain conditions.

It is normal to use a database as a concrete example of the central information source, so the concurrency control use the terminology used in databases. The concurrency control in asymmetric broadcast environments were first introduced by [7] when Shanmugasundaram et al. presented the APPROX algorithm. They proposed to decrease the level of the *correctness criteria* in order to get a good performing solution. A relaxed correctness criteria allows more transaction to be committed, but the consistency can not be guaranteed any more.

Later approaches have found inspiration in various database technology, such as real-time, distributed and conventional databases. The concept of OCC (Optimistic Concurrency Control) was developed in the early 80's by Kung et al. ([8]) and was later implemented as a basis for most of the concurrency control protocols in broadcast based environments.

The researchers constantly try to increase the performance and scalability, and new improvements are suggested each year. The newest scientific paper in the area is by Guohui et al. [9] and presents the concurrency control protocol EOCC. In the simulation results presented in the same paper, EOCC outperforms FBOCC from Lee et al. [10]. When other protocols are proposed, they usually only compare

the proposed protocol with old protocols. Also, different authors use different environment parameters in the simulations and it is therefore difficult to compare their performance out from the information given in the papers.

The protocols are many, and some are very complex. They possess various qualities and are special adapted for different environments, which makes comparison even more difficult. In order to structure and simplify comparison, some means to categorize the protocols based on their qualities is needed.

Several surveys have been written within the area of mobile databases and mobile transactions. Barabara [11] wrote a survey in 1999 about mobile computing and database. It deals with amongst others *data dissemination* and *data consistency*. The data dissemination introduces the basic of the broadcast disk etc. and the *data consistency* part review a few concurrency control protocols which are old by now. Many new protocol suggestions have been development the last 7 years since this survey.

Several surveys are even older, and others do only briefly touch the topic of cyclic broadcast and concentrate more on other topics of mobile computing (such as location awareness).

2.3 Claim and Project Definition

The environments we look at are asymmetric broadcast environments. By that we mean all networks where the server can reach all the clients with one message (broadcast) and the clients have less bandwidth up to the server than down from the server (asymmetric). The most common type of such a network is a wireless network. We will therefore use wireless networks as basis for most of our research, but the results are still applicable for other asymmetric broadcast based networks.

The project description states that we will make a survey, simulation model and a framework for categorizing. And although considerable amounts of work and money have been put into the research of this topic, we should also try to find improvements and new approaches.

None of the existing surveys cover the latest work in the topic of data management and concurrency control in broadcast based asymmetric environments. Fairly many approaches exists for both data management and concurrency control, and for people new to the topic it is difficult and a lot of unnecessary work to get a good overview of the topic. Therefore we mean a survey for this topic is useful.

To get better in-depth understanding of the protocols and to compare their performance, we made a simulation model. We implemented four concurrency control protocols and several techniques for data management. The performance of these protocols have not been compared in earlier work, so we made several simulations to find more out about the performance in various environment settings. We also present the source code to the simulation model, the implemented protocols and a basic framework which can be used as a base or platform to implement new protocols.

We also offer a qualitative characterization framework which gives a terminology to describe area of use and qualities of the protocols. The framework is helpful when choosing a protocol for a specific environment.

2.4 Report Structure

The third chapter introduces asymmetric broadcast environments and gives an explanation of the asymmetry. The focus is on the special characteristics and challenges, especially for wireless network and its clients. The last part presents some examples of data dissemination in broadcast environments.

Fourth chapter is about data management, and covers the research work that has to do with the structuring of the broadcasted data. Common terms such as latency, tuning time and broadcast disks are explained, and topics such as caching, index and data dissemination are reviewed.

Concurrency control is introduced in the fifth chapter. It defines three characteristics in order to categorize the protocols according to their qualities and suitability for various environments. The characteristics are later used in a framework, presented in chapter seven. The last sections review and summarize concurrency control protocols proposed for asymmetric broadcast environments.

The simulations are presented in chapter six. The structure of the simulation platform is described, as well as the protocols we implemented and the tests we executed.

The qualitative characterization framework is described in chapter seven, and defines criteria to describe the qualities to either a concurrency control or data management technique.

Chapter eight quickly present, discuss and evaluate the work we have done and our findings. The conclusion is in chapter nine, and it presents our final thoughts and suggest future work.

3 Introduction to Asymmetric Broadcast Environments

Many networks have the possibility to broadcast, that is the ability to send a message from one peer to all the others local peers. But the most interesting broadcasting networks in these days are wireless broadcast networks because of the huge increase of wireless capable handheld devices.

Wireless networks can be viewed as normal broadcast networks that have some additional constraints. So by focusing on wireless networks, the solutions can also be applied on normal broadcast networks. The solutions might be be too restrictive and therefore slow, but it should be easier to adjust to less restrictive than more restrictive. Therefore wireless networks will be given special focus in the rest of this report.

This chapter will introduce asymmetric broadcast environments by focusing on its characteristics and the special considerations that must be taken because of them. Chapter 3.1 introduces wireless networks and the special constraints that this kind of networks have. In chapter 3.2 we write about how bandwidth are allocated in asymmetric networks. After that, in chapter 3.3 we discuss handheld devices, and the limitations these special low power devices put on the broadcast system. Chapter 3.4 specifies characteristics and challenges in asymmetric broadcast environments. Followed by 3.5, where the data dissemination types are explained. And it is all rounded off by chapter 3.6 where some typical examples of broadcast environments are mentioned.

3.1 Asymmetric Bandwidth

Asymmetric bandwidth means the transfer rate from the server to the clients (downlink) is much bigger then the transfer rate from a client to the server (uplink). The downlink:uplink ratio is typically in terms of 4:1 to 10:1 depending on the environment. This asymmetry is quite common, partly because it fits good with many applications and partly because broadcast networks are inherently like this.

Most applications transmit a small amount of data, like a request, in the upload direction and receives a larger amount of data in response in the download direction. This is the case for many of the traditional network services and the bandwidth is therefore often divided in a way to fit this environment. This is especially the case in networks where the bandwidth is shared among both the uplink and the downlink. This applies to networks where the same media is used to send data in both directions, wireless communication is an typical example of this but it also applies to networks where the same wires are used for both directions. In these networks it is possible to allocate more bandwidth to communication in one direction than the other, for instance one timeslot for sending data and 9 timeslots for receiving. Also, having one server and many clients leads to asymmetry. The clients must be careful not to swamp the server with requests. [12]

If a broadcast network has the total upload rate equal to the total download rate, the bandwidth will still be asymmetric in one sense. The server can reach all the clients with one message, while the clients must share the same bandwidth among them. The upload bandwidth per client is therefore much less than the download bandwidth, hence asymmetry.

An environment in which clients have no backchannel is an extreme example of network asymmetry. Less extreme examples include wireless networks with high-speed downlinks but slow uplinks (e.g. cellular), and home Internet connection through ADSL.

3.2 Asymmetric Broadcast Networks

Asymmetric broadcast environments have limited upload bandwidth (because of the asymmetry). The clients can be either battery powered or have an permanent power connection. Battery powered devices are typically for wireless networks, and will be further described in the next section. For battery powered devices it is important to reduce the amount of network traffic to save power. The upload link is already reduced, so it is the downlink, or amount of access to the broadcast channel that must be reduced.

Two typical examples of asymmetric broadcast networks are cable TV and satellite.

3.3 Wireless Networks

Wireless networks are the most interesting type of broadcast networks, because of the high increase in number of users the recent years as well as its challenging limitations such as limited bandwidth, frequent disconnections etc. In addition, the wireless networks are inherently broadcast based. Data sent to one peer can in general be sensed by all the surrounding peers. That is, the core structure of the network makes the cost of a data broadcast to be exactly the same as the cost of sending a normal data package.

Even though it is self evident, we must mention for the sake of correctness, the ad hoc networks are not considered. They are constructed between peers “met by chance” and do therefore not contain any form for servers with big processing abilities. The non ad hoc wireless networks on the other hand are controlled by a base station who can reach all the peers through one broadcast message. This report will use the general term *wireless network* when in fact *non ad hoc wireless networks* is meant.

The wireless networks are different from wired networks in several areas. The transmission medium is more unreliable and have a much bigger bit-error rate. This unreliability can lead to frequent disconnections, for example when an external object blocks the signals. [13]

The bandwidth is physically limited by a finite number of frequencies. New techniques to modulate and compress the signals may be found, but the wireless bandwidth will continue to be a scarce resource [14].

3.4 Handheld Devices

Since wireless networks in general have lower bandwidth than wired networks, it is typically used in environments where it is unpractical to use a wired connection. In these environments battery powered devices are the main contributor. Even though the performance and network capacity on these devices have increased greatly the last years, the battery capacity is still almost the same as it was 10 years ago. This calls for methods to conserve power in all ways possible to maintain a good operation time while using the devices.

Most new consumer electronic use Lithium-ion batteries. This because of their long life cycle and high specific energy and energy density, and the fact that these batteries do not have the memory effect seen in other kind of batteries. Even though there have been some advancement in battery technology, there are new trends with smaller devices and requirement of low weight. This puts the capacity of the batteries at roughly the same capacity as they was 10 years ago. Even though both the weight to energy ratio and the volume to energy ratio has doubled the last 15 years [15], the main development has been done from the middle of the 90s to the early years in the 2000. The development has now flattened out and there is no reason to expect any radical change in the capacity of Lithium-ion batteries, most research are focused on finding new types of batteries and especially liquid fuel cell batteries.

There are many power draining components in the handheld devices. Display and storage is responsible for a big part of the total power consumption, but this is not affected by the broadcast traffic in any way so we can just ignore these factors. The factors that are affected by broadcast traffic is the usage of CPU power, and of course the usage of the network controller. While it can be argued that the access to storage can be affected by the broadcast the increased power consumption form that is so small that it can be ignored (especially when solid state storage is used).

The Intel produced PXA270 processor [16] is a widely used processor in PDAs which can run on various frequencies to save battery power. A high load on the CPU will cause it to use more power and a low CPU load will cause it to use less power. A PXA270 CPU clocked at 520 MHz uses 747 mW in active mode and 222 mW in idle mode. If it is possible to avoid any processing of data and put the CPU into 13 MHz idle mode (15.4mW) or maybe into standby mode (1.7mW, with the LCD controller turned off) the battery time would be greatly improved.

WLAN interface is quite common in wireless devices. A WLAN client device can reduce it's power consumption by turning off the WLAN device while waiting for the data to be broadcasted. The Philips BGW211 WLAN SiP [17] is a new 802.11g device that takes aim to cut down the power consumption of using WLAN. This chip is a fully integrated WLAN device so there will be no additional power consumption by external circuits for processing or amplifying. In receive mode the chip consumes 400 mW (802.11g), while standby mode consumes less than 2mW. Given the factor that we have a 1/200 ration in power consumption we can conclude that this is a good way to conserve power. An important factor is the power cost of sending data that is very high. In addition to the 400 mW needed to receive data there is a cost of 600 mW (802.11g at 15 dBm) for sending data. And the cost of sending greatly increases when the distance increases.

Mobile devices are more prone to disconnections than other devices, partly because moving out of area wireless networks cover and also because it will be disconnected to conserve power. This creates a need for special caching methods to take in consideration this special behaviour.

3.5 Characteristics and Challenges in Asymmetric Broadcast Environments

Most asymmetric broadcast environments consists of wireless networks with handheld devices as clients. This environment has the following characteristics and challenges:

- Wireless broadcast networks are characterized by:
  - Frequent disconnections
  - Low bandwidth
  - Asymmetric bandwidth where upload is expensive
  - Server can send broadcast messages to all the clients at the same cost as sending to one client
- And the following is the characteristics of the handheld clients:
  - Limited battery power
  - Limited processing power
- The data dissemination applications have the following properties:
  - Many read-only transactions compared to update transactions
  - One (or more) central information source(s)
  - Potentially very many clients

So the challenges with data dissemination is to :

- Limit the use of upload bandwidth
- Avoid battery consuming operations (CPU and wireless components) on the handheld device
- Avoid overloading the server
- Handle frequent disconnections

And at the same time as addressing the above points keeping the performance as good as possible. Performance is measured in how quick the broadcast items can be accessed, and how many transactions gets committed.

These characteristics are for wireless broadcast networks with battery powered handheld devices as clients. Other asymmetric broadcast environments have some of the same characteristics but are in general less restrictive. The clients can for example be continuously connected to a power outlet, and do therefore not need optimization to reduce the power usage.

3.6 Dissemination of Data in Asymmetric Broadcast Environments

The traditionally way of data dissemination is on-demand [18], also called pull based [12]. The clients requests the data they want and the server responds with the data. It is simple, but do not scale with a large number of clients. The waiting time for a data item can be very long if there are many transactions waiting for different data items. Also, the risk of the server being overloaded will increase with the number of clients.

In the broadcast approach [18], also called push based [12], the information server periodically and continuously broadcasts data items to the clients. If a transaction is waiting for a data item, it will wait for it to appear on the broadcast and then fetch it down. No matter how many clients there is, the same bandwidth and processing at the server is required, which is indeed very scalable.

To find the correct data item quickly, it is important with good data management. The data can be indexed so it is easy to find and power can be saved while waiting for the right time to read the item on the broadcast channel. Cache techniques can also be used to greatly decrease the latency. This is explained in detail in the next section, Data Management.

If the data items are allowed to be updated (which it is in practically all cases) and the clients have some requirements for consistency or integrity of the data, then concurrency control must be used. Consistency comes in many degrees [19] which is explained in more detail in chapter 5, Concurrency Control. The level of consistency is defined by a correctness criteria. The concurrency control makes sure all data is consistent according to the correctness criteria. Many protocols are suggested for optimizing performance. The major ones are also presented in chapter 5 Concurrency Control.

3.7 Examples of Data Dissemination Applications

And old example of data broadcast is Teletext where televisions with a special decoder (most TV's have that built-in now) can receive text info from the spare line of the vertical synchronization blanking line. This enables the opportunity to show text pages on a TV. Pages consisting of 40x28 characters [2] are broadcasted together with the TV-signal. Memory was a very expansive part in the time when teletext started to be implemented, so to keep the device cost down the decoders only had space to store one page (1 kbyte for B/W, 2-4 kbyte for colour) at the time. A teletext system may consist of many hundred pages of text. These are sent out in a cyclic chain [1] where it might take from 5 to 30 sec to get the requested page. This can be seen on as a storage in the air, because the devices only have a limited amount of memory and therefore have to wait until the data is broadcasted to be able to show it.

There are also other examples like ATIS (Advanced Traveller Information Systems) where information are broadcasted to inform about important events that might have interest for a big number of people, like information about traffic jams or detours. Stock market tickers are also an good example of data dissemination. There are a finite number of different stocks on a stock market, in most markets ranging from a few hundred to a few thousand, these can be propagated to all clients instead of having the clients make a connection and request the price for each stock. Of course there is a need for methods to optimize broadcast for these kinds of environments, techniques that are discussed in the later chapters.

Other examples of applications are weather information, auction, sensor networks etc. in general environments where a huge number of clients need to read the same data.

4 Data Management

Data management (DM) in broadcast environments consists primarily of choosing an efficient method for broadcasting data. How this is chosen depends on what kind of clients there is in the network, and on the characteristics of the data that is to be broadcasted. This provides the basis for concurrency control, which is discussed further in chapter 5. Many of the mature solutions already describe how caching is to be done to improve performance or they may describe the caching as a part of the protocol. In addition it might be possible to gain extra performance by compressing blocks of data. This is often left for further study when a new protocol is described.

First we give a introduction to the area of data management. In chapter 4.2 we explain different means of data access. Chapter 4.3 shows different ways to disseminate data to the clients. Chapter 4.4 describes different methods of indexing. In chapter 4.5 different caching methods are discussed and in the end chapter 4.6 quickly explains how compression may be used.

4.1 Introduction

In data management we have two very important measurement of performance, namely latency and tuning time [4]. Latency defines the total amount of time the client application need to wait for the data after a request. This number might be quite high in broadcast based environments and client side caching of often used data items is often used to try to keep the latency as low as possible. With other words, latency includes the time that is needed to actually receive the data, in addition to the time that is needed to locate the data. If the client actively listens to the broadcast or not, while waiting for the broadcast of a specific item, does not affect this value.

Some of the proposed methods requires synchronized clocks, this could be a challenge in the earlier days of computers. But now quartz based clock technology is cheap, and most devices have components for it, [20]. This opens the possibility for the usage of timing in algorithms.

Each dissemination period is called a broadcast cycle, this is the time it takes to broadcast all items in the database. And the broadcast content is called broadcast (bcast). The smallest logical unit of a broadcast is called a bucket. Broadcasts may also be divided into larger units like pages. Broadcast time is the time it takes to broadcast one bucket.

Tuning time defines the time it takes to locate the data. That means the time a client have to listen to be able to locate the data. For battery powered devices, this parameter is very important, since the parameter is the main factor for how long the battery power will last. The tuning time will affect the latency in some way or anther, but they are only indirectly linked to each other. Illustration 4.1 Shows the relationship between tuning time and latency. In environments where the devices have a low cost on the usage of power the latency is the important part. But on battery powered devices the tuning time combined with the time needed to fetch down the data is of greater importance.

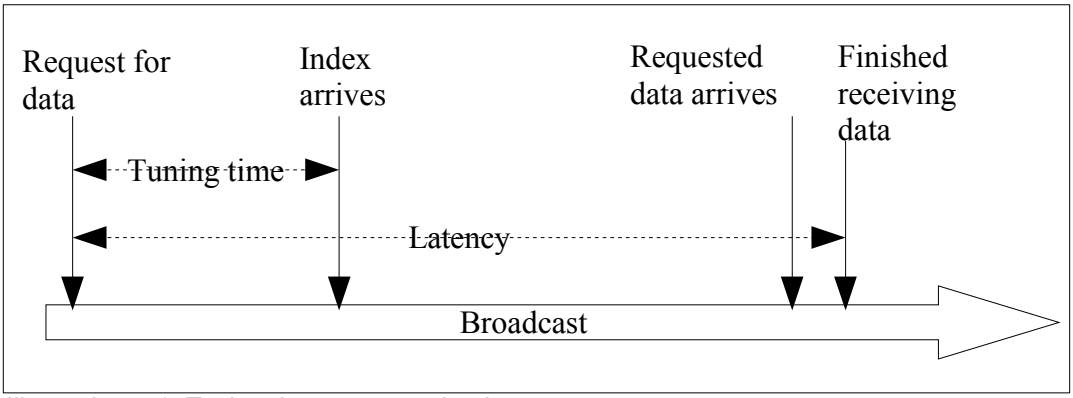


Illustration 4.1: Tuning time compared to latency

Hot data is defined as data that is useful for a numerous number of clients, while cold data is data that is needed by only a limited amount of clients, or no clients at all.

4.2 Data Access

There are three different ways a client can access data in an asymmetric broadcast based environment. One might be better for one kind of access patterns, but another may fit better for another. We here describe different methods of data access that are possible to use in asymmetric broadcast based environments.

4.2.1 Push [5]

The data is sent out to the clients without any need for communication back. This is an extreme example of an environment where the client has severely limited upload bandwidth or no upload capacity at all. Some traffic information systems works in this way.

For push based environments to be efficient, good knowledge about the access pattern of the clients is required so that the broadcast data can be arranged for optimal performance.

The advantage with push based data access is that it scales exceptionally good. The load on the connection and server will be the same, totally independent on the number of clients.

In handheld devices push based communication can help to greatly increase battery time, since sending data use a lot more power than receiving [21]. And the power needed for sending data greatly increases with the distance, due to the need for signal amplification [22].

4.2.2 Pull

Pull based environments are environments where all data is sent out solely based on requests from clients. This will typically result in a lower latency than push based methods up to a certain threshold. It might be done in two ways, either by broadcasting it on request or sending it on a dedicated channel to the client that request it. So if no requests for data are made no data will be sent.

While this works quite good in traditional RPC (Remote Procedure Calls) environments it does not scale very well. With many clients the bandwidth needed for the data requests can fast exceed the connection limit. And the load on the server can fast become a bottleneck since it has to keep track of a great number of connections.

4.2.3 Hybrid

Hybrid solutions are solutions where the hot data is broadcasted while cold data have to be fetched using a pull based method. This is a combination of both push and pull based communication and can result in better performance. But good algorithms are needed to determine whether the data is hot enough to be broadcasted. The requests are sent to the server on a dedicated backchannel. The response can be sent in two ways. Either it can be sent directly to the client using a own channel. This can result in a very low latency, but with lots of clients it will also result in high load on the server, and usage of much bandwidth. The other way is often called on-demand, where the requested data gets queued to be broadcasted. By requesting data that will not appear on the broadcast for a long time the latency can be reduced. But this again raises problems with how large a queue there should be on the server, a too large cue can result in a to long latency. But a too small queue can result in that the server will have to drop requests when the queue is full.

Acharya et. al proposes IPP (Interleaved Push Pull) [12] which uses a threshold for how many requests a client can send for missed data. So only data that would otherwise have a long waiting time, is requested. When a request is received, the request is appended to the queue if it is not already there. A predetermined limit decides how much of the broadcast that can be sent out, due to incoming requests. If this limit is set to 0% we have a standard push based broadcast.

4.3 Data Dissemination

Broadcast techniques generally depend on two important factors, the latency and the tuning time. This will often be a trade-off, by reducing latency you increase the tuning time and vice versa. This often makes comparison between different broadcast techniques hard, since different methods perform better in certain environments. When a method is made it usually targets a very specific environment, and to make it perform good in another environment requires modification to the protocol.

While most techniques only describes how to do broadcast using one channel it is often possible to extend the specifications to also be applicable in environments using multiple broadcast channels. In most cases multiple channels can just be multiplexed into a single virtual channel and be treated in just the same way as a single channel.

4.3.1 Scheduling

Scheduling is how to decide what data to send first, some data might have a higher priority and have a greater cost when it is delayed.

There are many approaches for scheduling, that means that different methods calculate using different costs. Some use linear scales, but other take into account that waiting for the first elements might have way higher cost than it would have when waiting for the 20th element.

4.3.1.1 Linear

The cost for waiting for an item is equal all the time. Wither this is the first item, or the twentieth does not matter. This is very simple to implement, and requires no calculations at all to decide the schedule.

4.3.1.2 Polynomial

In [23] it is proposed to use polynomial functions to calculate the cost of having a cache miss. This means that when the cost is calculated a miss on the first few elements will weight more than a miss on the 20th element.

The paper also does an theoretical analyzation on the performance of four common scheduling methods:

- The random algorithm
- The Halving algorithm
- The Fibonacci (golden ratio) algorithm
- The Greedy algorithm

While this might increase the waiting time for some elements the responsiveness of the clients will increase.

4.3.2 Broadcast Methods

The standard approach to data broadcast is using a flat pattern that is broadcasted in cycles. Given the data “A”, “B”, “C”, “D” and “E” it will be broadcasted like this “A B C D E A B C D E A ...”.

In [24] M. H. Ammar and J. W. Wong use Markovian Decision Process (MDP) to prove that an optimal method of data dissemination in a push based environments is a cyclic one.

4.3.2.1 Broadcast Disks[5]

Acharya et al. propose using multiple in air disks, spinning at different speeds. The fastest disk contains the hottest data, while cold data is placed on a slower spinning disk.

From [5]:

The algorithm has the following steps (for simplicity, assume that data items are “pages”, that is, they are of a uniform, fixed length):

1. Order the pages from hottest (most popular) to coldest.
2. Partition the list of pages into multiple ranges,where each range contains pages with similar access probabilities. These ranges are referred to as disks.
3. Choose the relative frequency of broadcast for each of the disks. The only restriction on the relative frequencies is that they must be integers. For example given two disks, disk 1 could be broadcast three times for every two times that disk 2 is broadcast, thus, *rel_freq(1) = 3*, and *rel_freq(2) = 2*.
4. Split each disk into a number of smaller units. These units are called chunks (C_{ij} refers to the j^{th} chunk in disk i). First, calculate *max_chunks* as the Least Common Multiple (LCM) of the relative frequencies. Then, split each disk *i* into *num_chunks(i) = max_chunks / rel_freq(i)* chunks. In the previous example, *num_chunks(1)* would be 2, while *num_chunks(2)* would be 3.
5. Create the broadcast program by interleaving the chunks of each disk in the following manner:

```
01 for i := 0 to max_chunks - 1
02        for j := 1 to num_disks
03                Broadcast chunk C_{j,(i mod num_chunks(j))}
04        endfor
05 endfor
```

This gives us an efficient method for broadcasting data in a cyclic manner, and it is quite simple to implement, but fine tuning of the disk speeds needs to be done to make it fit different environments.
In hybrid environments broadcasts disks might be seen on as in air cache that removes the need for a pull request.

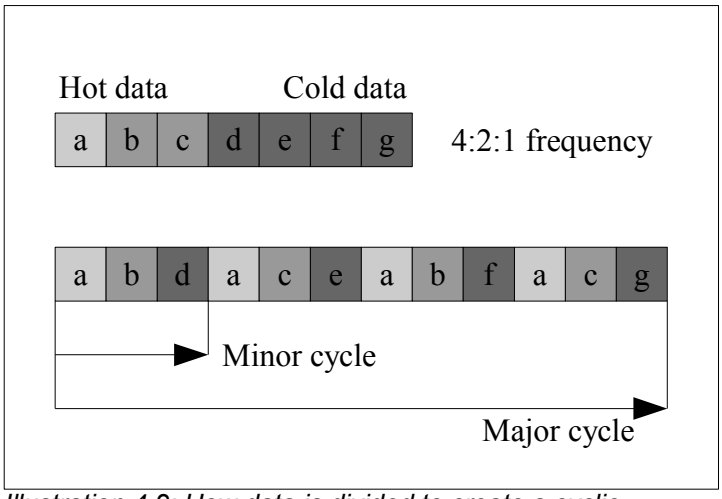


Illustration 4.2: How data is divided to create a cyclic broadcast in broadcast disks

4.3.2.2 A Binary Approach (BNB) [6]

Broadcast disks suffers from an empty slot problem, where some of the timeslots will be unused if the broadcast is divided into parts that can not be equally distributed. A proposed solution to this is GBNB where the data is divided up in a different manner.

GBNB have some requirements to be implementable:

1. Each data item is in one page of the same size.
2. The client population and their access patterns do not change.
3. Data is read-only.
4. Clients make no use of their upstream communications capability.
5. When a client switches to the public channel, it can retrieve data pages immediately.
6. A query result contains only one page.
7. The server broadcasts pages over a single channel.

The disk frequency in GBNB is calculated based on the client access probability thus giving us an optimal disk frequency where there are no empty slots. This gives an method that performs better, or equally good as broadcast disks in all cases, but puts some extra restrictions on the implementation.

4.3.2.3 Multiversion

The approach contains several versions of a data value, so if two items are updated and only one of them were read before it was updated, the other item can still be read consistent by choosing the old value. In other approaches restart would be the only option.

The drawback is obviously the huge overhead of maintaining several values for each item, and in broadcast environments the overhead is especially associated with broadcasting many times of extra data (although techniques to optimize exists).

Making multiple versions might be done in two ways, either distributed horizontally or vertically [25]. This might affect how efficient it is to do compression on the data and should therefore be weighted before choosing what method to use.

4.3.2.4 TC-AHB [26]

In TC-AHB (Time critical adaptive hybrid broadcast) the broadcast is divided into two parts, the periodic broadcast and the on-demand broadcast. This is dynamically adjusted to fit the actual user access frequency distribution. The server performs an classification of all the data at the end of each broadcast

cycle to determine which ones to broadcast in the next broadcast cycle. These decisions are based on the access distribution from the last broadcast cycle and bandwidth cost for sending each item. The items that are expected to save bandwidth by being periodically broadcasted instead of broadcasted on demand are scheduled for broadcast during the cycle. To prevent starvation there will always be left a small amount of the bandwidth for on-demand broadcast.

While the method does not specify any special indexing method, there is need for one so that the clients can decide wither the data will appear on the broadcast, or if they have to request it to be sent on-demand.

This works good, but when the broadcast program gets more and more accurate to the clients need the less information the server will get about the user access distribution. To cope with this problem, it is possible to cut the broadcast of the items for a short period of time, so that the clients will request the hot data again, and we will have new values for the classification of the data items.

4.3.3 Invalidation Lists

Invalidation lists are lists that are sent out from the server to indicate that a data value have changed, and that the clients need to download the updated version. While it is possible to send out the invalidation lists at any time during the broadcast, many methods choose to do it at the start of it.

[27] does some research on invalidation lists. The performance of invalidation lists are measured in different environments, using cache, multiversion and SGT (Serialization-Graph Testing).

Invalidation lists prove to be efficient in environments where there are a lot of updates and rapid changing data.

4.3.4 Propagation [28]

Instead of sending lists over what data that has changed, propagation simply sends the new value of the changed data. This way the clients will stay close to the steady state, since they do not have to wait for the invalidated data to reach it. But it will also be a waste of bandwidth to propagate data that is not used by any clients, or very few. The updates may be sent at any time during the broadcast sequence. While propagation works very good in environments with few updates it suffers from the problem that in update intensive environments the propagation can take up too much of the broadcast bandwidth.

Data propagation my be scheduled in many different ways, dependent on the environment some might be more efficient than others. [28] Proposes the following three propagation methods.

4.3.4.1 Server Offset

Propagates the versions that the server think is mostly used. With increasing noise the server assumption drift more and more away from the real client situation.

4.3.4.2 Slow Disk

Only propagate the items on the slow disk, where it would impose a long waiting time for the data to be broadcasted again.

4.3.4.3 Threshold

The data is propagated only if the next sending time is larger than a given threshold. This threshold is given as a percentage size of the broadcast size. The method is highly dependant on what part of the pages that gets updated most frequently. If the most frequently accessed pages also are the most updated ones the best threshold is at 10%. With other update patterns different thresholds provide the optimal propagation level.

4.3.5 PA [29]

PA (Predeclaration and Autofetching) uses an preprocessor in the beginning of each broadcast cycle to determine which data that is needs to fetch. This results in a bit longer minimum delay, but gives a lower average delay.

PA outperforms both IM and MA when the update rate is high or the transaction length becomes longer than five data items. This because PA will always fetch the data within two broadcast cycles totally independent of transaction length or how many of the data items that have been updated. But PA suffers in environments where there are few updates, since it has to wait until the start of the next broadcast to start the prefetching. To solve this problem PA^2 was made, this is the same as PA, except that it works asynchronous and starts the prefetching immediately and therefore will in some cases be able to complete within less than one broadcast cycle.

4.3.6 SGT [27]

In SGT (Serialization-Graph testing) the client maintains a copy of the serialization graph with all the read operations. Updates are integrated into the local copy. Read operations are accepted as long as they do not create a cycle in the local graph. On the server side there might be a different kind of concurrency control, such as two-phase locking.

The usage of SGT requires the serialization-graph and control info to be sent at some time during the broadcast cycle. Then typically at the beginning of each broadcast.

SGT requires quite a lot of storage space at both the clients and the server since the SG have to be stored to be able to check against it. While it in most cases is possible to simplify the SG there must still be reserved enough space for the full graph in special cases where that might be necessary.

SGT requires quite a lot of processing at the client side to check the graph, this might be a limiting resource and have therefore to be be taken into consideration.

4.4 Index

Indexes are used to give a description of the placement of the broadcasted data. While there are numerous different kinds of indexes, some seem to perform better in broadcast environments than others.

Caching of the index is a good way to cut down on tuning time, but when data is cached it is important that there are methods for invalidation if the structure of the broadcast should change.

To make the data easily accessible it is important to have a structure of the data. Dependent on what kind of indexing method that is going to be used, different requirements for the index structure applies. The simplest one is the one dimensional structure, where the index is just stored in a sequence one after another. Multilevel indexes arrange the data in structures that makes it easy to find. A good example of this is the commonly used index tree where a structure of leaf nodes makes up a big tree. An important part when making these kinds of trees is to get the structure balanced. And some of the the following theoretical formulas here in this chapter rely on a balanced tree for the theoretical values to be correct.

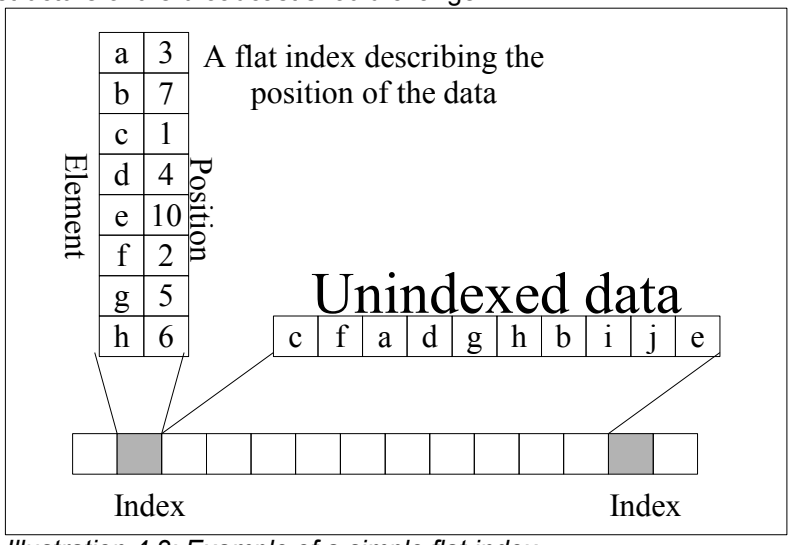


Illustration 4.3: Example of a simple flat index

Indexes may appear in two different forms, clustered and none clustered [20]. In clustered indexes similar kinds of data are grouped together but in none clustered they are spread around on the whole index. As might be expected, none clustered performs worse or in best cases equally good as clustered ones.

4.4.1 Latency Optimal [4]

This is not an index method, but rather a technique the other index methods can be compared against. It defines the latency optimal method of data broadcast. This is done by keeping the broadcasted data as small as possible. Therefore all indexes are omitted.

As given by the name of this method, this makes the latency very small, and it is the simplest method since it in reality is what you get when you do not implement any indexing method at all. Even though this might work good in some environments the constant listening on the network traffic would often drain to much power from battery powered devices. But it is a good method to compare how close the techniques come to the optimal latency.

The expected latency in the Latency Optimal method is calculated by taking the time it takes to broadcast a single data item and multiply that with the number of items that are being broadcasted. This gives us the total time to broadcast all the data. If a client requests a totally random data there is an equally chance for it to be any of the items, and since the items are equally spaced on the broadcast the average time it takes

will be the item in the middle of the broadcast. So therefore we divide on two to find average latency for an item. And we have to add the time it takes to receive the data.

$$\text{Latency} = \frac{\text{Broadcast time} \times \text{Database size}}{2} + \text{Time to recieve the data}$$

Illustration 4.4: Latency of the Latency optimal method

Since the client has to listen to the broadcast at all time to find an item the tuning time is calculated in the same way as the latency, except that the tuning time do not include the time that is needed to receive the data.

$$\text{Tuning time} = \frac{\text{Broadcast time} \times \text{Database size}}{2}$$

Illustration 4.5: Tuning time of the Latency Optimal method

4.4.2 Tuning Optimal [4]

This method defines the shortest possible tuning time for an environment where the data access is totally random. To accomplish this an index describing the layout of the pages are broadcasted. This index is broadcasted in the start of every broadcast cycle. This gives a high latency, since all page requests need to wait for the index broadcast to see when to fetch the data. Each broadcast have a pointer to the next index which is the first element in the broadcast. Therefore the expected tuning time will be half of the time it takes to broadcast one data item plus the time it takes to receive the index. Just like latency optimal this is a method created to do performance evaluation on other methods, and is in most cases not a practical method since it gives a very high latency.

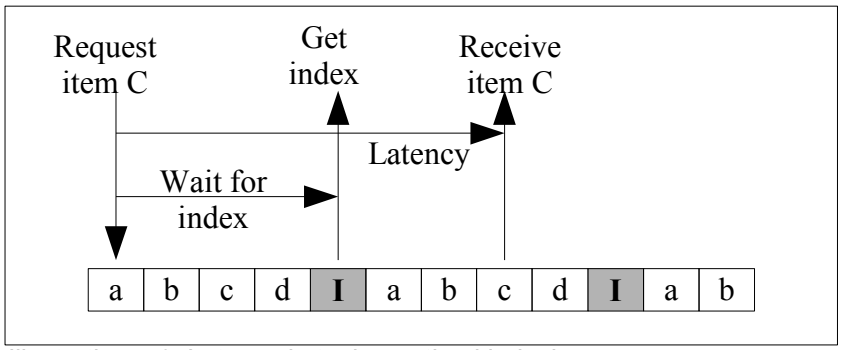


Illustration 4.6: Latency in tuning optimal indexing

To calculate the latency we know that on average we will be in the middle of the broadcast when we start looking for an item, so we first have to wait half a broadcast cycle to receive the index, then we on average have to wait another half broadcast cycle before the data is sent on the broadcast and a broadcast cycle is the same as the broadcast time multiplied with the database size. We also have to add the time it takes to receive the index, and the time it takes to receive the data.

$$\text{Latency} = \text{Broadcast time} \times \text{Database size} + \text{Time to recieve the index} + \text{Time to recieve the data}$$

Illustration 4.7: Latency of the Tuning optimal method

$$\text{Tuning time} = \frac{\text{Broadcast time}}{2} + \text{Time to recieve the index}$$

Illustration 4.8: Tuning time of the Tuning optimal method

4.4.3 (1, m) [4][20]

In (1, m) indexing, a full index is broadcasted once for every m items. This method keeps both tuning time and latency in mind, at the extra cost of a few broadcast slots for broadcasting duplicates of the index.

The procedure for the (1, m) indexing method is as following.

- Tune into the broadcast.

- Fetch when the next index will be broadcasted.
- Turn off receiver until the time for index broadcast.
- Read index and get the time for when the page is to be broadcasted.
- Turn off receiver until the time for the page broadcast
- Retrieve the required data.

(1, m) is a simple and good method, but the overhead from sending the full index each time is high. But in environments where the index occupies only one bucket or less, this performs very well.

To calculate the expected latency we first find the time it takes to receive first index. Then we add up the time we have to wait before the item appears. Again on average the data will appear after half a broadcast cycle and we have to add up the indexes that will appear in between too.

$$\text{Latency} = \frac{1}{2} \times \left(\text{index} + \frac{\text{size of broadcast}}{\text{m}} \right) + \frac{1}{2} \times \left((\text{m} \times \text{index}) + \text{size of broadcast} \right) + \text{Time to recieve the data}$$

Illustration 4.9: Latency of the (1, m) method

Tuning time = probe time + time to follow the index and download it

Illustration 4.10: Tuning time of the (1, m) method

The tuning time for (1, m) is calculated as in Illustration 4.10 First we need a probe to find the next index, then the client can stop listening on the network traffic until the index appears and we have to follow the tree in the index and download the item that describes our requested data.

$$Optimal\, m = \sqrt{\frac{\text{Size of data}}{\text{Size of index}}}$$

Illustration 4.11: Optimal value of m

It is also possible to calculate an optimal value of m using the formula in Illustration 4.11. This formula is made by solving the formula in Illustration 4.9 with respect to m.

4.4.4 Distributed Indexing [4][20]

When the data is ordered in a directory structure there is only need to broadcast the index for the upcoming data. So what distributed indexing does, is to improve (1, m) indexing by cutting down on the replication of the data.

Distributed indexing may take three forms:

- Non-replicated distribution
- Entire path replication
- Partial path replication

The different types describes how much of the directory structure is replicated. Where the non-replication and the entire replication are both two extreme cases of replication. The full method is explained in Imielinski et al. [4][20]

While this method is fairly complicated to implement, it is very efficient and performs better than (1, m) in all cases except when the index is small enough to fit within a single broadcast bucket.

4.5 Caching

Caching of data on the client side may often prove efficient on the performance of the clients. Different DM methods may make it possible to turn of the data receiving interface and just use cached data for periods of time, to save battery power. The efficiency of caching often depend heavily on how rapid data changes. If data rarely changes or does not change at all a big cache would greatly improve performance. But for caching to work efficiently there is need for methods to decide the validity of the data.

The lack of bandwidth and the high disconnect rate in mobile environments makes page based caching unsuitable since the cost of replacing a full page is to high. That leaves us to two types of caching, attribute based and object based. Where an object is a collection of one or more different attributes that has some kind of a relation to each other. A hybrid version where both object and attribute based caching is used is also possible. Which one that performs best greatly relies on the environment it used in.

Conventional caching methods often keeps track of the clients and sends invalidation messages when the data changes. This is not suitable/feasible in these environments where the clients may disconnect often and freely.

There is an possibility to use lease based caching. But that leaves us with the problem of deciding the lease time. If the time is too short, it would lead to wasted bandwidth. But then again if the time is too long it will give a greater chance of invalid data. Leased based caching is therefore not practical.

Cache have warm up time where the cache is not stable and values that should have been cached might be swapped out with with another value that have a lower access rate. This state is called the warm up phase. Different caching methods have a different length of warm up time and in environments with very rapid disconnections this warm up time might play a role in the access time. When a client reaches the state where the cache is stable it is called the steady state [12].

Acharya et al. [5] describes that it is wasteful to frequently broadcast pages that are highly likely to be in the clients cache. When the client has reached the steady state. While this is totally true it is also something that is hard to avoid in mobile environments where we have rapid disconnection and the clients might roam to areas out of coverage.

4.5.1 General Methods

A wide range of caching methods has been proposed for mobile environments. Some are specific for a given kind of broadcast others are more general methods that can be implemented and extended for use in broadcast schemes and transaction control that has yet to include caching in the model. In the following sections a few of these are presented.

4.5.1.1 PIX [30]

Idealized algorithm, and implementation is hard, if not impossible, as it requires exact knowledge of the access probability of the client. A PIX value is calculated (Illustration 4.12) for each element in the cache by dividing the frequency of broadcast with the access probability. It woks by always replacing the cached data with the lowest PIX value, something that requires a comparison between each element each time new data is received on the broadcast. This puts a severe load on the clients who perform the caching, this makes PIX a more theoretical caching method.

Probability of access over frequency of broadcast

$$\mathrm{PIX} = \frac{\mathrm{P}}{\mathrm{X}},\ P = \textit{frequncy of broadcast},\ X = \textit{access probability}$$

Illustration 4.12: Calculation of the PIX value

4.5.1.2 LIX [28]

Since PIX is often looked on as a none implementable algorithm LIX was developed as an approximation of PIX. It is very similar to the well known LRU, but instead of just removing the last link of the chain PIX calculates a cache hit value between the multiple disk and removes the one with the lowest cache hit value and puts the new link in the chain corresponding to the disk it belongs to.

There will be one chain for each disk on the broadcast. These chains will contain the entire cache memory. When a new item arrives, a calculation will be done for the last item on each chain and the new item. The item with the lowest value will be replaced. And the new item will be put in the chain corresponding to the disk it resides on. If the number of broadcast disks are reduced to one LIX will be reduced to LRU.

A small problem with LIX is that none updated items flush to the end of the chain faster. Since a item is moved to the top of the chain when it is updated.

4.5.1.3 PT [30]

As another theoretical caching method we have PT. This is a prefetching method that takes into account the time it will take before the data appears on the air again. For this to be possible a PT value has to be calculated for each element each time an element is received on the broadcast. These calculations does of course put way to much load on the clients and are therefore not really implementable in a real life situation.

There is also a need for exact knowledge about the access probability for all the elements on the broadcast.

4.5.1.4 Tag-team [30][31]

Tag-team caching uses a 180 degree algorithm to always cache the items that it takes the longest before they appear on the broadcast again. Therefore cutting the cost of a miss. Illustration 4.13 Shows and example of an broadcast with four elements and a cache size of two elements, the items that will arrive again soonest on the broadcast is swapped out with the one that it takes the longest time before it will appear again. This method is made solely for reducing the penalty when a cache miss occurs, it does not improve the hit ratio compared to standard caching methods.

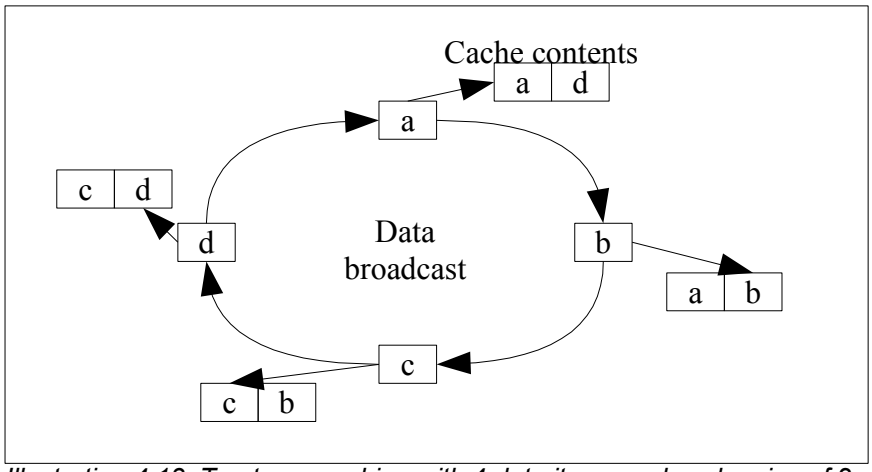


Illustration 4.13: Tag-team caching with 4 data items and cache size of 2

4.5.2 Real Time

Real time cache methods are the cache management techniques made specifically for real time environments, where transactions have to be performed within a given deadline to be accepted.

4.5.2.1 LDF [32]

LDF (Largest First Access Deadline Replaced) is created for use in real-time environments where transactions have a deadline they have to reach. It works by constructing a deadline for the latest time an data item can be read to be able to reach its deadline. LDF defines many ways to do this calculation, but performance wise they all give very similar results. While this seems simple it requires to know quite a few parameters in advance:

- The exact number of data items
- The sequence of the items
- The data id
- The soft data-deadline

Although LDF seem to perform way better than PIX in environments with deadline restrictions it seems like the performance improvement decreases when the cache size goes up and also we can see that the performance of PIX get way better when it is applied on a broadcast containing multiple broadcast disks.

4.6 Compression

Compression divides the data into blocks that becomes compressed by using different kinds of methods. What method and the size of these blocks is dependent on the environment its to be used in. Usually the main factor that affects the efficiency rate of caching is the randomness of the data. If many data values contains the same data a greater rate of compression can be achieved.

The problem with compression techniques in these environments is that it is often a bit CPU intense to perform the calculations for compression [25]. So this might actually be a trade off between network receiving time and CPU time.

5 Concurrency Control

Concurrency Control (CC) protocols are important to maintain the consistency of a database and to make sure the clients reads consistent data.

In the middle of the 90's, small wireless devices started to be common, and asymmetric broadcasted environments became a bigger topic. To deal with, amongst others the limited upload bandwidth, some new concurrency control protocols were searched for. Many of the concurrency control protocols from the traditionally database processing were then adapted to the asymmetric broadcast environment. Therefore it is important with some knowledge of concurrency control in traditional databases. The first section in this chapter focus on the concurrency control protocols for traditional databases that makes the foundation for many of the concurrency control protocols in the broadcast environments.

Section 5.2 explains the importance of concurrency control and give examples of problems without concurrency control and how concurrency control can prevent these. The third section 5.3 defines three characterisation criteria of concurrency control protocols for asymmetric broadcast environments. They describe the environment the concurrency control can be used in and are later used in the characterisation framework.

The sections from 5.4 to 5.14 review various concurrency control protocols suggested for the environment in question. Only a selection of the protocols will be reviewed, and also only the interesting features will be explained. The reviews focus on the functionality and techniques used for better performance. For a complete understanding of the protocols, the reader is referred to the cited scientific papers.

5.1 Introduction to Concurrency Control and its Protocols

A database management system (DBMS) is a collection of software and hardware that support commands (operations) to access the database [19]. A simple DBMS would execute each operation atomically and sequentially, but most of today's DBMS can execute operations concurrently. That is, many transaction can be committed concurrently as long as the final effect is the same as a sequential execution. Operations can be executed concurrently if they operate on different data items.

Concurrent transactions that do operate on the same items will in most cases lead to inconsistency in the database. The operations causing the inconsistency is called *conflicting operations*, and conflicting operations cause a *conflict*. The task of concurrency control is to avoid conflicts in order to ensure the consistency of the data. Various concurrency control protocols ensures different degrees of consistency. The degree of consistency is often (and so also in this report) described as a *correctness criteria*. The concurrency control protocols will accept different kinds of concurrent transaction depending of the correctness criteria. Correctness criteria will be described in more detail in section 5.3.2.

The two major techniques for concurrency control are lock based and non-lock based. Lock-based was used in the first DBMS and it ensures consistency by putting a lock on the data items accessed by the operations. A lock reserves a data item and is not released before the transaction is committed, which in turn efficiently avoids conflicting operations and an inconsistent database (more on lock based can be read in Bernstein et al., [19]). The approach is simple, but has many drawbacks, where some are deadlocks and poor support for real time transactions [19]. Especially in asymmetric broadcast environments, lock based protocols have many drawbacks . They require extensive bi-directional communication which uses too much of the uplink capacity. A large population of clients will also overload the server with read locks [7]. In addition, locks would be held for a long time because transactions are longer in broadcast environments (the clients have to wait for the correct broadcast item). And in case of disconnection some locks could end up not being released.

Because of these drawbacks and the emerging of new database types such as real time, distributed, and object oriented databases, new non-lock based techniques were invented such as multiversion, timestamp, serialization graph and optimistic concurrency control. The non-lock based techniques are the best protocols for broadcast environments [7], and will therefore be more detailed described in the coming sections. Multiversion and serialization graphs are introduced in the previous chapter so they are only shortly mentioned here.

5.1.1 Timestamp Ordering

A unique timestamp is assigned to each transaction. It is indeed problematic to generate an unique timestamp in a distributed fashion, but some techniques are already described in chapter 3.11 in [19]. The timestamp can for example be generated by a counter (possible a synchronized clock) plus a unique number given to each client (or just a random number assuring uniqueness even if two transactions were

started at the same time). In many cases it is not necessary to have an unique timestamp, so a broadcast cycle number can be used.

Dynamic timestamps were introduced by [33]. The principle is to adjust the timestamps in such a way the transactions will be consistent.

5.1.2 Serialization Graph Testing [19]

A serialization graph is used to detect conflicts in the history of transactions, and is a graph consisting of transactions and their operations. Most of the papers presenting new concurrency control protocols use serialization graphs in formal correctness proofs. If no cycles occur in the graph, the history of transactions is serializable.

In *serialization graph testing*, the serialization graph is maintained at the server with the following modifications. The graph does not contain very old transactions (because of space and speed considerations) and the graph contains active non committed transactions in addition. This modified graph is therefore called *stored serialization graph* (SSG).

Transactions are only accepted if SSG is still acyclic. The drawbacks are the substantial length of the SSG (although techniques to shorten it exist) and the computing power required. Although this approach is not suitable for the asymmetric broadcast environment at all, an approach using SSG does exist for broadcast environments [34].

5.1.3 Certifications and Optimistic approach

The concept of certifications was first developed by Thomas in 1979 [35]. The certification approach accepts all operations and checks for conflicts later. It must at minimum check when the transaction commits [19]. This approach was also developed by H. T. Kung independently in 1981 [8], but he called it an optimistic approach, which indeed it is. All operations are executed with the optimistic idea that no conflicts will arise. First later, maybe as late as the time of commit, the protocol will check for conflicts.

Transactions in OCC have three phases [8], read, validation and write. In the read phase, all operations are scheduled. Write operations are written to a local temporary storage. The validation can be done several times but must at least be done when the transaction commits. The transaction is checked for conflicting operations up against earlier committed and/or active transactions. If the validation succeeds, the transactions commits successfully and is written to the database. In case of conflict and rejection, the transaction is aborted or restarted. In order to discover conflicts early, validation can be done on the first operations before the last operation has started. If a conflict is discovered, the transaction can restart and avoids wasting resources on the remaining operations.

OCC were later split by Härder [36] into *Forward Validation* (OCC-FV) and *Backward Validation* (OCC-BV). OCC-FV turned out to be suitable for real time databases because of its ability to prioritise transaction by letting a high prioritized transaction be committed before low prioritized transactions. OCC in general turns out to be a very good approach for wireless networks. As earlier stated, the data dissemination applications has most read-only transactions and the asymmetric environment has a low uplink transfer rate. OCC works best in environments with few conflicts, which is the case when most transactions are read-only. Secondly, OCC can be used without sending any information on the uplink before validation. The other approaches requires sending of lock requests etc. So OCC reduces the use of the uplink if not too many conflicts arises.

OCC-BV validates the transaction up against the already committed transactions. The set of operations in the validating transaction is compared with the write set of committed transactions in the time space from the start of the validating transaction. It is not necessary to compare up against older committed transactions.

OCC-FV validates the transaction up against the active transactions (those which have not yet committed). Forward validation gives the opportunity to choose which transaction should be restarted. Either the validating transaction or the conflicting active transaction must be restarted. The choice can be done based on some prioritizing scheme. In addition, forward scheme generally detects and resolves data conflicts earlier than backward validation, and hence it wastes less resources and time [10].

But even conventional OCC has it drawbacks in broadcast environments [37]. For a client it takes a long time to know if a validation has failed. The restart of the failed transaction is therefore consequently delayed. A serious conflict that leads to a transaction abort can only be detected in the validation at the server. Then transactions destined to restart after the first operations are fully executed. This wastes time, resources and bandwidth. Also, the ineffectiveness of the validation process at the server leads to many unnecessary transaction aborts.

5.2 Some Concrete Examples of Concurrency Control

In order to illustrate the need for concurrency control, this part will show some examples of database inconsistency caused by concurrent transactions. First some serial executions are given and then the typical write -read conflict are illustrated in several forms.

To express a read operation or write operation on an item **x** by a transaction T_1 we write $T_1r(x)$ or $T_1w(x)$ respectively. $C(T_1)$ means T_1 is committed successfully into the database set. A sequence of operations is called a history and is denoted with H. For example, H_1: T_1:read(a), T_2:read(a), T_1:write(a). The order of the operations illustrates the order of the execution in time.

5.2.1.1 Two examples of serializable executions

H_1: $T_1r(a)$, $T_2w(b)$, $C(T_2)$, $T_1r(b)$

There is no difference if T_1 reads **a** before or after T_2 writes **b**. The value **a** is independent of the T_2 transaction, hence no conflicts. The same example is illustrated once more in Illustration 5.1 in a different way for clarity.

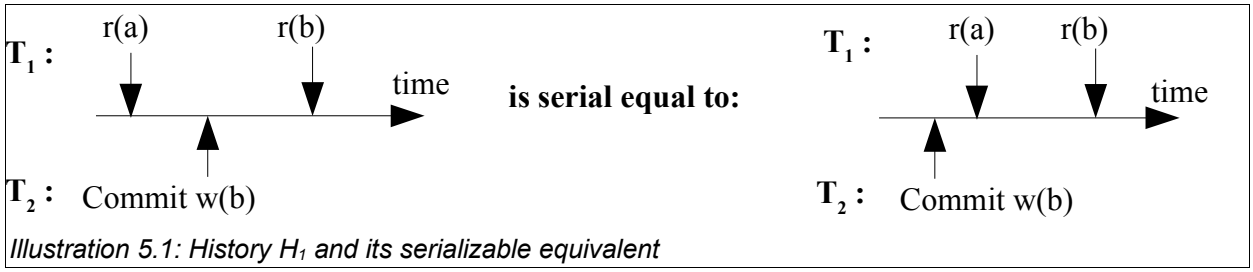


Illustration 5.1: History H_1 and its serializable equivalent

An alternative example illustrating the same point is a transaction where item **a** is written by T_2 instead of item **b**, like this:

H_2: $T_1r(a)$, $T_2w(a)$, $C(T_2)$, $T_1r(b)$

H_2 will also be serializable. It is clearly seen that there are no conflicts, because both values read by T_1 are old. T_2 do not update the value of **b**, so the transaction is still valid. See also Illustration 5.2 for clarity.

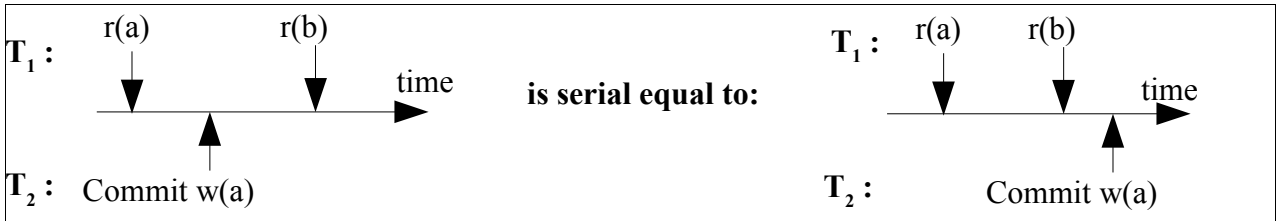


Illustration 5.2: History H_2 and its serializable equivalent

The lesson learned is that if an operation is moved in time within an interval where the operation do not conflict with any other operations on the same item, then the same serial execution is achieved. In the last example, $T_1r(b)$ can be moved before and after $T_2w(a)$, without changing the result, because the two operations are independent and not conflicting.

5.2.1.2 Write – Read conflict example

The write-read conflict is very common and easy to misunderstand. If a value **x** is read by T_1 right before it is updated by another transaction T_2, most would think it is a conflict. But even though the value read by T_1 is old, the consistency is not necessary threatened. First when T_1 accesses another item updated by T_2, the consistency is broken.

We give a history H_3 where transaction T_1 reads data item **a**. Then T_2 updates and commit data item **a** and **b**, and finally T_1 reads **b**. Illustration 5.3 gives a nice overview of the operation order.

H_3: $T_1r(a)$, $T_2w(a)$, $T_2w(b)$, $C(T_2)$, $T_1r(b)$

T_1 will read inconsistent values, because the value to data item **a** is old and the value to data item **b** is new. The inconsistency arises because items **a** and **b** are related, and a serial execution is not possible. The data item **a** and **b** are assumed to be related because they were accessed in the same transaction. A transaction is a series of operations on items where one operation is dependent of the other.

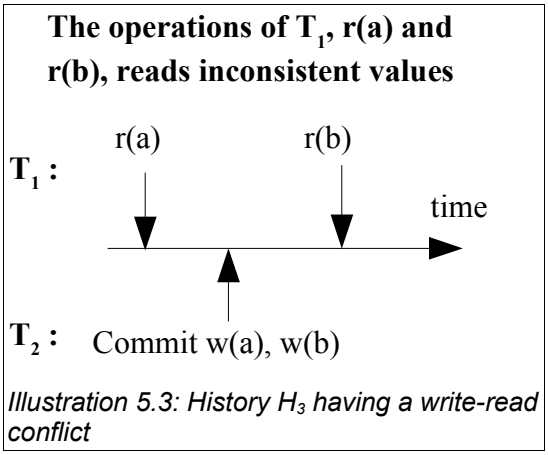


Illustration 5.3: History H_3 having a write-read conflict

Solutions in case of H_3, is to read the old value to **b** (which can be done with multiversion), or read the new value to **a**. If the last option is chosen, it is not enough to only read item **a** again, because the next operation (in this case read item **b**) is (potentially) based on the value of the current operation (in this case item **a**). So the whole transaction must be restarted.

If $T_1r(a)$ was not the first operation, it would strictly be enough to only restart from the point of where the conflict was found. The normal approach is to start the transaction from the beginning again, but since the first operations before the conflict still are valid, it is not necessary to do those again. We name this approach *partial restart* and investigate it later in the end of this chapter.

5.2.1.3 Dependent write-read conflict example

H_4: $T_1r(a)$, $T_2w(a)$, $T_2w(b)$, $C(T_2)$, $T_3r(b)$, $T_3w(c)$, $C(T_3)$, $T_1r(c)$

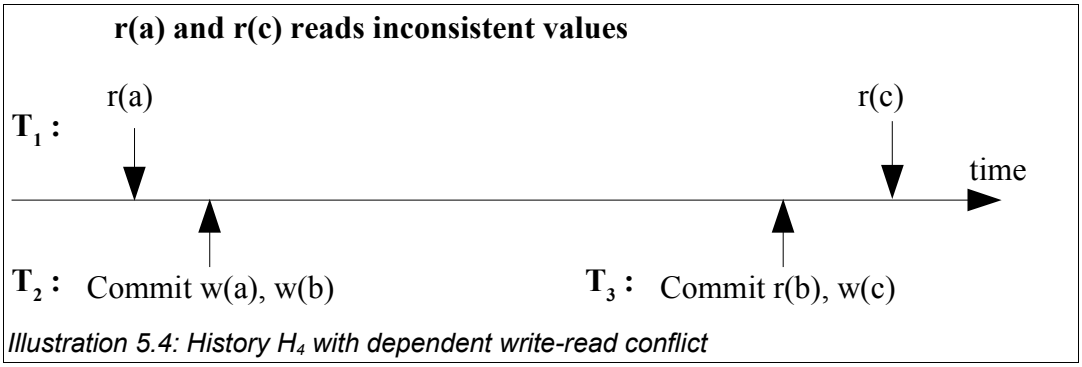


Illustration 5.4: History H_4 with dependent write-read conflict

In H_4, Transaction T_2 makes a dependency between item **a** and **b**, and T_3 gives a dependency between item **b** and **c**. So when T_1 wants to read **c**, the value is inconsistent with the value read for **a**.

Various concurrency control protocols deal with these inconsistency problems in many different ways. In last example the lock-based protocols would deny $T_2w(a)$ until T_1 committed. Optimistic protocols would allow all operations, in the optimistic hope of no inconsistency, until the validation of T_1. They would then discover the inconsistency and restart T_1. A third solution is to use the old value for **c** in the last operation $T_1r(c)$ when the inconsistency is discovered. Multiversion, shortly described in previous chapter, offers this opportunity.

5.3 Concurrency Control Protocols Characteristics

The concurrency control protocols for broadcast environments have many different characteristics. This chapter tries to point out three important differences that will make it easier to see which environment they are suitable for. The protocols have support for:

- client update transactions (or not)
- real-time transactions or not
- serializable consistency or other weaker correctness criteria

The next three sections will discuss these properties, and later in the report, these properties are used to categorize the concurrency control protocols.

5.3.1 Client Update Transactions Versus Client Read-only Transactions

All protocols assume there are update transactions. If not, the database set will be the same the whole time and no concurrency control is needed. These update transactions can be committed by the server or the

clients. But not all protocols allows the clients to perform update transactions. In fact, many applications do not need the clients to update data, but want the clients to be read-only peers. Example of such is a weather forecast or traffic information service. The clients only need to read information. On the other hand, an expanded version of the application is also thinkable, which lets the clients send the local weather or traffic conditions to the server for updated information.

There are two important differences between the server update transactions and client update transactions. The server usually updates an item based on information from a sensor (temperature sensor, traffic sensor, etc.). The value will then be committed independently of other values (blind write). Secondly, the server can quickly communicate with the database, while the clients must wait for the correct data item on the broadcast channel. Therefore server transactions do not need special adapted concurrency control but can use traditional locking based concurrency control. Still, the data set must be kept consistent, also for server transactions.

Because almost all the protocols implement server update transactions, we will sometimes use the expression *update transactions* when *client update transactions* is meant. If server update transactions is meant, it will be specifically specified. We will use the expressions *read-only protocols* and *update protocols* to represent protocols which support respectively only read-only and both (read-only and update) transactions.

It is natural to believe the read-only protocols will perform better in client a read-only environment (because it is special adapted for this environment) than an update protocol in a client read-only environment (because it is not special adapted for the environment). But most update protocols implement the read-only transactions separately from the update transactions because in most cases it is more efficient to use special algorithms which take advantage of the knowledge that the transaction only read ([38], [39]). Therefore we believe most read-only protocol implementations also can be expanded to an update protocol, and consequently the performance will be equal for the update protocols and read-only protocols in a client read-only environment.

If so, then supporting *client update transactions* is viewed as a stronger characteristic than not update compatible. That is, the update protocols are an extension of read-only protocols.

5.3.2 Real-time Versus None Real-time

Real-time databases have time constraints associated to some of the transactions. They must be committed within the limit of this time constraint which can be in form of a deadline. The transaction correctness in real-time is defined as meeting its time constraints and using data that is absolutely and relatively timing consistent [40]. If the deadline is exceeded, the transaction is usually aborted, because there is no need to commit it. *None real-time transactions* do not have a deadline and can still be correct even though it is delayed for a long time before it is committed. In that case the latency will be very bad, but it is always a wish to complete the transaction.

The real-time support is given by prioritizing the transactions by giving the transactions priorities based on, amongst others, their deadline. When a conflict is found, one or more transactions must restart. In a real-time system, the transaction with shortest upcoming deadline might be committed while a transaction with long time before the expiry of its deadline is restarted. Of course many more factors play a role when choosing which transaction to restart, such as how many transaction must be restarted if one other is allowed to continue, and a transaction's probability to complete within its deadline. Another fact that should be considered is that the cost or restart is very different for an update transaction from the server versus an update transaction from a mobile client.

Many applications needs real-time support because they are real-time in nature [10]. For instance in stock trading a delay of a transaction past a time constraint can lead to financial loss or opportunity loss.

The real-time support in the proposed protocols are of various degree. We categorize them as real-time *supporting* or *optimizing* if the scientific paper proposing it claims so.

Real-time is the stronger criteria because by simple *first-serve-as-first-come* prioritizing, the effect is as an environment without real-time. Some concurrency control techniques is not able to support real-time, but they may be faster in non-real time environments. It might therefore be a waste to use a real-time supporting protocol in a non real-time environment. Another possibility is that real-time protocols perform just as well as the non real-time protocols, even in non real-time environments. The reason for this assumption is to believe that the cost for real-time comes for nothing. Not prioritizing in a real-time protocol will not degrade performance.

5.3.3 Various Correctness Criteria

The correctness criteria decides how strict the transactions should be validated before they are committed, in order to maintain the database. The correctness criteria can be expressed in many ways, and often terms as *serializability*, *consistency* and *currency* are used.

In one sense, the clients view on the database can be described as snapshots of the central database. However, the states of a snapshot on one client does not necessarily correspond to the same on another client. If they do, they are *mutually consistent* and belong to the same *consistency group*. *Currency* is another term commonly used, and it refers to to how current or up-to-date a data set is. [41].

The most normal notion of correctness is *global serializability* [19] or just *serializability* as it is usually called. Under many circumstances, the costs of enforcing serializability is too high. This is especially true in real-time databases where it is often better to produce a useful result on time with a non-serializable schedule than producing it too late with a serializable schedule. [40]. In applications where the serializability is not important, it is preferable to have many non-serialized transactions committed then few serialized transactions. Therefore some researchers search for other more relaxed correctness criteria than serializability.

The next parts will review some correctness criteria and their characteristics.

5.3.3.1 Serializability

Serializability makes sure the results produced by the interleaved execution of a set of transactions should be identical to one produced by executing the transactions in some serial order [19]. This is one of the strictest correctness criteria, and do therefore allows fewest transactions.

Another issue, is how to implement the serializability control. The pursuit of guaranteeing serializability may lead to the rejection of many serializable transaction because of a bad implementation.

Most papers argue strongly for or against the use of serializability in asymmetric broadcast environments. Those against mean it is too expensive to use serializability because it will lead to many unnecessary restarts [7], [42], [43]. They mean many serializable transactions can be rejected in the pursuit of guaranteeing serializability. And because serializability is a global property, all the concurrent transactions should be executed as in some serial order, which is difficult to achieve.

The other front claims it is not expensive to use serializability, and they also have results that supports this assumption [10], [44], [45]. Any weakened correctness criteria should still implement currency, and then it is not so easy to find a protocol which performs better [29]. Also, they argue with the importance for certain application types to have a serializable schedule. Such as trading where a buy/sell trade will be triggered to exploit the temporary pricing relationships among stocks. From the trader's perspective, the inability of maintaining serializability may lead to important financial consequences [46]. For instance, if the users who submitted multiple read-only transactions communicate and compare their query results, they may be confused [38].

5.3.3.2 Update Consistency

Shanmugasundaram et al. [7] introduced a correctness criteria called *update consistency* by [47] and *external consistency* in [48]. It ensures the consistency of all update transactions and each read-only transaction is serializable with respect to the subset of update transactions it reads from. That is, it ensures mutual consistency on the central database and the data read by the clients, and currency on the data read by the clients. To decide if a history is update consistent is a NP-problem. So to implement the criteria, they made an algorithm which accepts a proper subset of *update consistent* histories.

5.3.3.3 Single Serializability and Local Serializability

Huang et al. [42] looked at *single serializability* and *local serializability* in search for a relaxed correctness criteria. Single serializability is when all the update transactions and any single read-only transaction are serializable. Local serializability, on the other hand, is when all the update and read-only transactions on one client side are serializable.

This is only a small selection of correctness criteria. Global serializability is the stronger criteria, but protocols supporting it may have reduced performance compared to more relaxed criteria, in environment where global serializability is not needed.

5.4 Datacycle [3]

Update transactions:	Yes for the Old Datacycle and No for the new Datacycle
Real-time:	No
Correctness criteria:	Serializability

Table 5.1: Overview of Datacycle characteristics.

Datacycle is presented in [3] and [46] and is the first concurrency control technique published in the literature aimed at broadcast environments, according to our knowledge and to Shanmugasundaram et al. in [7]. It is not aimed for asymmetric networks, but high-speed broadcast networks. It supports client update transactions, do not support real-time and uses serializability as correctness criteria (see Table 5.1).

Later Lee et al. [49]proposed a new protocol with similarities to the one just mentioned, and therefore called it Datacycle too. From now on we will call the Datacycle protocol presented in Herman et al. for the O*ld Datacycle* and the newest one for simply *Datacycle* or *new Datacycle*. For the sake of correctness, Datacycle was also mentioned and used in relation to R-matrix in [7], but we do not refer to that one.

5.4.1 The Old Datacycle

The Old Datacycle was aimed at high-speed fibre networks for propagation of broadcast data, specifically telephone network data such as name-address translation. The broadcast technique was used because it scaled good. It was developed to be able to handle the huge data requests over 10 000 transactions per second on databases with 10 millions of items. By letting the clients itself do the queries the server would not be overloaded. It uses a certification based concurrency control algorithm and it ensures that all transactions executing at clients and the server are globally serializable.

Each broadcast cycle has a beginning and a start and no updates are committed in this period. That way the data set in one broadcast cycle are always consistent and read-only transactions can be done locally as long as they only access data within one broadcast cycle. In the telephone network it is no problem to complete a read-only transaction within one cycle, but for handheld devices it is difficult because of limited cache size, processing power and battery power.

For update transactions a certification based protocol is used.

5.4.2 The New Datacycle

In a paper presenting concurrency control protocol BCC-TI [49] the new Datacycle is also presented for comparison to BCC-TI. And since BCC-TI is a read-only protocol, so is the Datacycle. The essence consists of committing client read-only transactions locally, without communication to the server. If a read-only transaction can complete its execution with no conflicts with committed transactions during its execution, then it can commit autonomously (alone and independent) without the need to abort any active transactions.

The difference to the Old Datacycle is that the new is not required to execute a whole transaction within one broadcast cycle. *Control information* (CI) is sent for each broadcast cycle such that a transaction can span multiple cycles. CI contains the write set to the transactions committed during the previous broadcast cycle, and the CI is broadcasted right before the start of a new broadcast cycle. The CI is a kind of invalidation list, which was reviewed in chapter 3.

When a transaction is committed, it is not broadcast at once in the current broadcast cycle. The committed transactions write set is recorded and put in the next CI, which is broadcasted before the next cycle. The new committed values are broadcasted in the following cycles. In other words, the CI contains the write set, which is the data items with new values in the coming cycle. That means each broadcast cycle only broadcast consistent data, and by looking at the CI, the client can determine if its read set is consistent into the next broadcast cycle.

The client (read-only) transactions checks its current read set up against the broadcasted write set (CI). If any conflicts are detected, the transaction aborts. As long as a transaction is executing/active, it is checked up against all broadcasted CI's. The transactions can start at whatever time it wants in the broadcast cycle (obviously) and stop when it wants. There is no need to wait for the last CI, because that is only needed if some elements in the next cycle is read.

The advantages of CI and the new Datacycle is the autonomously committing of read-only transactions, and partly solution to the late restart problem (late discovery of conflicts which leads to transactions destined to be rejected are not restarted before the final validation). The technique of using CI has been used by most new approaches, in order to process read-only transaction locally and get an early restart of conflicting transactions.

5.5 Certification Reports [50]

Certification reports uses a modified version of OCC and it uses conflict information to perform the validation and hence guaranteeing the serializability. The schedulers name is *WoundCertifier*. It supports update transactions and serializability, but have no considerations for real-time (see Table 5.2).

Update transactions:	Yes
Real-time:	No
Correctness criteria:	Serializability

Table 5.2: Overview of Certification Reports characteristics

The key to the protocol is the use of *certification reports (CR)*, which is information sent on the broadcast channel that the clients can use to validate its active transactions. The CR contains read and write set to recently committed transactions and the result of the validation (accept/reject). The CR is analogous to the CI.

It is required that the clients listen to the broadcast channel continuously during an active transaction. Read-only transaction are not mentioned especially, so they must be validated as all other transactions at the server (Although it looks like it is not necessary as long as the CR information is available).

Conflicts are discovered early, and Barbara claim 90% are discovered at the client side. This saved upload bandwidth and time, because an early restart can be done.

In [51] by Das et al., the server validation verifier *WoundCertifier* is replaced with *COREV* and *R^2COREV*. The replacements accept more transactions, but still maintains the serializability.

In addition the server validation answer is sent directly to the client through a dedicated back channel. Then the answer is guaranteed received (because of handshake) as opposed to reply on the broadcast channel which can be lost because of a small signal error. Also the client do not have to listen to uninteresting information about other transactions that is accepted or rejected.

They also conclude that if all the transactions were validated at the server, each would stand a chance of being re-ordered in case of conflict. But bandwidth, client power usage and scalability is wasted in such a case. If all transactions were validated at the client, the throughput would not be optimum. So there is a trade-off between performing the transaction validation on the server versus on the client.

5.6 Read-only Transaction Processing [52]

Pitoura suggests three ways of supporting read-only transactions at the client without contacting the server and still maintain the consistency. Real-time and client update transactions are not supported, as Table 5.3 indicates.

Update transactions:	No
Real-time:	No
Correctness criteria:	Serializability

Table 5.3: Characteristics overview of read-only approaches by Pitoura

Multiversion broadcast (covered in previous chapters) and *invalidation-based consistency* is suggested. The latter broadcast invalidation lists to the clients, so they can decide the consistency. Two approaches are suggested, namely *conflict serializability* and *invalidation-only broadcast*.

For the conflict-serializability method, both the mobile clients and the server have to maintain a copy of the serialization graph for conflict checking. It incurs high overheads to maintain the serialization graph. The integration of updates into the local copy of the serialization graph and the cycle detection may be too computation intensive for portable mobile computers.

In the invalidation-only broadcast, a read-only transaction is aborted if any data item that has been read by the read-only transaction is updated at the server, and it results in low concurrency.

5.7 APPROX, F-Matrix and R-Matrix [7]

[7] is, to the knowledge of Lam et al. [18], the first study on concurrency control in broadcast environments that uses a control matrix for concurrency checking. They introduce a polynomial time approximation algorithm named APPROX, as well as two implementations of it, namely F-matrix and R-matrix.

Update transactions:	Yes
Real-time:	No
Correctness criteria:	Mutual consistency and currency

Table 5.4: Overview of APPROX characteristics

Shanmugasundaram et al. argue for a simpler correctness criteria, and introduce *update consistency* in order to satisfy mutual consistency and currency. See Table 5.4.

The control matrix contains the broadcast cycle number for the latest committed transaction on the database items. The control matrix is broadcasted for each broadcast cycle, and the clients validates their transactions using the control matrix. The validation checks for conflicts between the latest committed transactions and the validating transaction. If a read-only transaction does not have any conflicts, it is committed locally in contrast to the update transactions which are sent to the server. For any conflicts, the transactions are aborted. The update transactions write to a local copy at the client before it is committed, such that the server traffic can be kept to a minimum. This approach is widely used by most of the protocols.

The control matrix is the predecessor to the CI (Control Information) used in most newer protocols. For F-matrix (Full matrix) it consists of a n x n matrix, where n is the number of items in the database. An element C(i, j) is the cycle number (timestamp) for the latest committed transaction. C(i, j) is the latest cycle number in which some transaction that affects the latest committed value of item(j) and also writes to item(i), commits.

In order to avoid huge cycle numbers in CI, the maximum number of cycles a transaction can last is defined. Then CI can contain the value of “cycle_number mod maximum_cycle_numbers”, and hence save space.

The server broadcasts the latest committed values of all data items in the beginning of each broadcast cycle. And at the end is a control matrix which help the clients to know if read-only transactions are valid. The server validates the transaction and updates the control matrix and the database set if the transaction is committed.

Instead of viewing each item as a separate entity, several items can be viewed as a group. R-matrix is the extreme case where all the rows in the F-matrix is combined into one row, thus all items are viewed as one unit. The advantage is the much smaller CI, but the disadvantage is poorer performance because of worse granularity.

The F-matrix has bigger overhead when sending the control matrix, but it accepts most transactions because of the fine granularity. The R-matrix has less overhead, but will therefore reject more transactions because of bigger granularity.

The major drawback is the maintenance of the matrix and the size of it when it is broadcasted. The considerably size can lead to long delays. Another drawback is the weakened correctness criteria, because several applications are dependent of the serialized transactions.

Their simulation results showed F-matrix as the better one in nearly all the tests, but they did only simulate client read-only transactions and not client update transactions.

An advantage with local read-only validation is the clients can validate with algorithms adapted to their correctness criteria. Different clients may have different currency requirements and even for a given client, there mar be different currency requirements for different data items. Since the invalidation of the cache at clients is purely local, the invalidation interval can be tailored on a per client, per object basis and the invalidation performed accordingly. Thus, clients with vastly different currency requirements can coexist in a broadcast medium without any need for extra communication.

The major drawback of this approach is the large overhead needed to maintain the matrix for concurrency control and conflict checking. The matrix will use up a substantial percentage of broadcast bandwidth especially when the size of data item is small or the size of the database is large. Therefore using such a large matrix is impractical in many real-life applications. Maintaining the control matrix also involves complicated processing at the server.

5.8 UFO,Update-First with Order [18]

UFO is a read-only protocol which implements serialization as correctness criteria. It has no sense of real-time and only the server can commit update transactions (Table 5.5).

Update transactions:	No
Real-time:	No
Correctness criteria:	Serializability

Table 5.5: Overview of UFO characteristics

A maximum number of broadcast cycles the transactions can last is defined. If a transaction lasts longer, it is dropped. This same principle was used by F-matrix and R-matrix in order to keep the data field for the broadcast cycle low.

UFO divides into two phases, that is execution phase and update phase. The transactions can span several broadcast cycles because of the rebroadcasting of the updated values. That is, the server updates are broadcasted right away. To ensure the consistency within one broadcast cycle, the items that are updated after they are broadcasted are re-broadcasted. The items are re-broadcasted at once they arrive, which

leads to a not constant size on the structure of the broadcast. With other words, the clients must listen to the broadcast channel the whole time. This issue is mentioned as feature work in the paper.

The UFO algorithm can maintain the serializability of update transactions at the database server and the read-only mobile transactions. The algorithm has minimal overhead and can be applied in different broadcast models.

Other than data re-broadcast, the UFO algorithm does not affect the basic mechanism of the underlying broadcast algorithm. Furthermore, an advantage of the re-broadcast is that every time a data item is being broadcast, it will be the most updated version.

If the data conflict is high, the number of re-broadcast would be significant and have much impact on the broadcast model. In this case, the UFO algorithm should be enhanced to minimize bandwidth required for re-broadcast. One possibility is to broadcast invalidation message for the update item instead of re-broadcast the update item itself in order to save bandwidth.

5.9 BCC-TI [44], [49]

Update transactions:	No
Real-time:	Yes
Correctness criteria:	Serializability

Table 5.6: Overview of BCC-TI characteristics

Lee et al. introduced the term BCC (Broadcast Concurrency Control), and propose the two new concurrency protocols BCC-FV and BCC-TI. Both use serializability as correctness criteria and has optimizations for transactions with a deadline (real-time). The protocols do only support read-only transactions from the clients (Table 5.6).

BCC-TI is a optimisation of BCC-FV. Both broadcast control information so the clients can validate transactions locally.

They focus on avoiding the late restart problem. And suggest avoiding sending the read-only transactions to the server for validation, because a read-only transaction do not have any writes, and will therefore not cause any read-write conflicts, based on the principle of forward validation. Write-read is the only conflict possible.

BCC-TI expands Datacycle, and use timestamps to order the serialization order, so a read-only transaction can precede an update transaction. It avoids unnecessary restarts.

5.9.1 BCC-FV

Broadcast Concurrency Control with Forward Validation (BCC-FV) is based on OCC-FV on the server side. It takes advantage of the fact that read-only transactions committed to the server will not cause any other transactions to restart because read-only transactions does not contain any write set. (Remember OCC-FV uses the write set to a committing transaction and checks it up against current active transactions). Therefore the read-only transactions only have to be validated locally.

Forward Validation is used to avoid the late restart problem, which is late discovery of conflicts and transactions destined to be restarted are not restarted before they sent for final validation.

The local validation concerns write-read conflicts with the write set to the committed transactions. Therefore the write set to the transactions committed in last cycle is broadcasted in the Control Information. If the write set in the CI intersects with the read set from the read-only transaction, it will restart. The late restart is partly solved and the read-only transactions can be processed/committed locally.

However, BCC-FV suffers from the unnecessary restart problem. Therefore an optimization of BCC-FV was also presented, namely BCC-TI.

5.9.2 BCC-TI

Broadcast Concurrency Control using Timestamp Interval (BCC-TI) uses timestamps to avoid the unnecessary restart problem. To easier describe the protocol, this terminology can be used:

- WS(U), write set of an update transaction U
- TS(U), final timestamp of U
- WTS(d), largest timestamp of committed update transaction that has written data object d.

When a transaction U commits, TS(U) is given the current timestamp and for all elements d in WS(U) the WTS(d) is set equal TS(U). That is, all items that is written to by the committing transactions gets its WTS set to the current timestamp. The TS(U) and WS(U) are recorded into the CI.

The broadcast consists of the CI followed by the data items with their belonging WTS.

At the client each transaction is given an initial timestamp interval [0, ∞> to determine if the transaction is valid. For each read operation of item d, the lower bound (LB) is set to WTS(d) if bigger then the current LB. The validation is done continuously as long as the transaction is active, by comparing the current read set against the write set in the CI. The upper bound (UB) is adjusted by comparing the current read set to the write set in the CI. If the read set intersects with an item in WS(U), the UB is adjusted to the minimum of UB current value and TS(U). After the adjustment, the LB must be compared to the UB. The transaction must be restarted if LB >= UB.

The protocol offers autonomy between the clients and the server, because they can work independently. It also offers flexibility by adjustments of the timestamps. The unnecessary restart problem is partly solved. These benefits comes with the cost of managing timestamps and broadcasting them.

In the end of each broadcast cycle is control information (CI) broadcasted. The CI contains the committed WS during the current broadcast cycle. Then the client can validate its read-only transactions and restart them at once when a conflict is discovered. If LB >= UB, a conflict is registered and the transaction must be restarted. Early data conflict is detected.

5.10 STUBcast, [42]

Update transactions:	Yes, but natively not from the server!
Real-time:	No
Correctness criteria:	Single and local serializability

Table 5.7: Overview of STUBcast characteristics

STUBcast supports update transactions at the client side, but do not natively support any kind of server transactions. This can easily be implemented by using a local two phase locking and performing the same procedure as with an update transaction from a client. STUBcast has no optimizations for real-time, but this is written to be a feature research area in the scientific paper. It introduces two new correctness criteria, namely single serializability (SS) and local serializability (LS) as shown in Table 5.7. The clients can commit read-only transactions locally.

STUBcast is a quite complex protocol and validates operations through the use of timestamps. Its performance compared with other protocols is not known.

SS ensures all update transaction and any read-only transaction to be serializable. LS requires all the update transactions in the system and all read-only transaction at one client side to be serializable. SS and LS is weaker but easier to achieve than global serializability, and they do guarantee the consistency and correctness at the server database.

STUBcast implements SS and LS by dividing broadcast operations into primary broadcast (pcast) and update broadcasting (ucast). The pcast broadcast all the database items with their value, while ucast broadcast committed transactions and is inserted into the ongoing pcast whenever a transaction is committed. The protocol consists of three parts, client side read only serialization protocol (RSP), client side update tracking and verification protocol (UTVP), and server side verification protocol (SVP).

RSPss accepts read-only transactions if and only if it confirms to SS (all update transactions and any single read-only transaction are serializable). Conflict cycles are avoided by maintaining a conflict array (CFA). The CFA is maintained with data from the ucast. If a read-only transaction wants to read an item that is in the CFA, an cycle is implied, and the transaction must be aborted.

UTVP records the write operations and other important information (such as read operations and timestamps) in a RECarray, and submits it to the server for final validation. The RECarray is maintained and validated with data from the update transaction and ucast. The exact behaviour is described in [42] but a simple policy is to abort when a written item occurs in a ucast and the timestamp to the ucast is bigger.

SVP validates the RECarray by checking the serializability to already committed transactions. If serializable, the data is written to the database and the items written to are all assigned the current timestamp. Then a ucast broadcast is prepared and inserted into the broadcast.

Together they guarantee SS or LS in all accepted transactions. A server timestamp is broadcasted with each data item. The server timestamp describes the time the item was last updated. RSP takes care of read-only transactions at the client side, UTVP takes care of update transaction from the client and SVP takes care of the incoming update transactions at the server side. SS or LS is decided by choosing RSPss or RSPls.

The simulations show in the paper are done with RSPss with only one client and one server. To simulate several clients they use a short *inter transaction time* (short time between each transaction). We mean it is important to have several clients when simulating update transactions because concurrent update transactions leads to conflicts. We understand one client as only one transaction is sent at a time. The next

is not sent before the first is received. Even if several transactions were sent concurrently from one client, the concurrency of the transactions will be different. One clients that starts transactions with a uniform time between them, will always remain uniform. By having several independent clients, the transactions can after a while be sent all at once (bursts) and sometimes none, like a real environment.

The simulation results are compared with no concurrency control, and gives results close to that when the transactions are shorter than 12. STUBcast is not compared with other concurrency protocols.

The mobile clients must listen to the whole broadcast during a transactions, because there exist no way to know when next ucast will come. This is also listed as a future optimization, and can be fixed by using a tradeoff of how quick the ucast must be broadcasted.

5.11 PVTO [37] [45]

PVTO uses OCC with partial validation and timestamp ordering, where PV represents the OCC part with partial validation and TO represents the timestamp ordering. PVTO supports client update transactions, uses real-time optimizations and uses serializability as correctness criteria (see Table 5.8).

Update transactions:	Yes
Real-time:	Yes
Correctness criteria:	Serializability

Table 5.8: Overview of PVTO characteristics

The new updated values are not broadcasted right away, but first in the following cycle after the CI is broadcasted. The CI states the rejected and accepted client transactions in previous cycle, the read set (*CT_ReadSet*) and write set (*CT_WriteSet*) of transactions committed and accepted in the last cycle, and finally *RTS(x)*, *WTS(x)* and *FWTS(x)* which is read, write and first write timestamps to data items x found in *CT_ReadSet* and *CT_WriteSet*.

As seen, the CI is quite complex and so is the rest of the protocol. Also, all (included read-only) transactions must be sent to the server for validation.

Each data item in the database is assigned a read timestamp, *RTS(x)*, and a write timestamp *WTS(x)*. They represent the youngest committed transaction that respectively read and wrote the item. Just like BCC-TI, each active transaction, T_A, is associated with a timestamp interval (TI) which is initialized [0, ∞>, TI lower bound, $TI_{LB}(T_A) = 0$ and TI upper bound, $TI_{UB}(T_A) = \infty$.

The timestamp ordering has the same function as in BCC-TI, so if the interval is shut out ($TI_{LB} >= TI_{UB}$) the transaction is restarted.

Timestamp makes sure unnecessary restarts are avoided. Partial validation detects data conflict on an early stage, thus avoiding a late restart.

The timestamps are dynamic. That means a transaction may be committed after another transaction, but because it read old values it is serialized before by dynamically adjusting the timestamps.

Each transaction, also read-only transactions, are transferred to the server for validation. This consumes more upload bandwidth and may swamp the server with requests. But in environments where performance in form of response time is very important, a small increase in performance is achieved with this method compared to FBOCC. This in trade with more consumption of upload bandwidth, client timestamp management and server processing (validation).

5.12 OCC-TI [43], OCC Based on Timestamp Interval

U. Lee et al. suggested this approach which supports weak consistency. Updates transactions are supported and no optimizations are made for real-time (Table 5.9). The approach uses timestamp interval with upper and lower bound to validate the transactions.

Update transactions:	Yes
Real-time:	No
Correctness criteria:	Weak consistency

Table 5.9: Overview of OCC-TI characteristics

The protocol uses the broadcast channel to announce accepted and rejected transactions. The protocol is similar to PVTO and BCC-TI but uses a weaker correctness criteria, and should therefore perform better. Unfortunately no performance is available, but the protocol is described good and verified formally.

5.13 FBOCC [10]

Update transactions:	Yes
Real-time:	Yes
Correctness criteria:	Serializability

Table 5.10: Overview of FBOCC characteristics

FBOCC has all the strong characteristics such as serializability, client update transactions and real-time optimization (see Table 5.10). It uses forward validation at the server up against active transactions, and backward validation at the client based on the information received from a small CI. In addition the server does a quick backward validation of client update transactions when the they are sent to the server for final validation. The client read-only transactions are committed locally.

The CI is broadcasted in the beginning of each broadcast cycle, and it contains the set of data objects committed (updated) in the last cycle and a identification (cycle timestamp). The new updated values are not broadcasted right away, but first in the following broadcast cycle, after the CI have “announced” their id.

The partial backward validation at the clients is a check of all current read operations (the read operations made at the client in one transaction) up against CI which contains the last broadcast cycle's committed write sets. If a conflict is found, the client transaction will be restarted. The read set at the client must be checked against each CI as long as the transaction is active. If the transaction is an update transaction, the write values are written to a temporary storage at the client, and the transaction is sent to the server for validation. In case of any committed writes after the client sends the transaction for validation at the server, it checks the read set when it arrives. This is called the final validation. At the end at the forward validation, the write values to the validation transaction T_V are checked up against the read set of the active transactions at the server.

The read-only transactions can be completely autonomously validated and committed by the client by checking against the CI constantly during its execution phase.

The update transactions must record the broadcast cycle they send the validation. By help of the partial validation at the server, the transaction is guaranteed to have no inconsistency up to the point of the last checked partial validation against the write set in CI. By sending the broadcast cycle timestamp, the server can do the final validation (backward validation) and validation (forward validation) against current active transactions. This is when the prioritizing for real-time can be performed.

The special optimizing for the server update transactions is to restart the server instead of clients. Because clients are expensive to restart. This approach can be implemented in other protocols as well.

The simulation results in presented in the proposal of FBOCC ([10]) only shows a comparison up against conventional OCC.

5.14 EOCC [53]

Update transactions:	Yes
Real-time:	Yes
Correctness criteria:	Serializability

Table 5.11: Overview of EOCC characteristics

EOCC is a variant of OCC and supports all the strong characteristics as shown in Table 5.11. Like the others, it processes read-only separately from the update transactions.

Guohui et al. separates severe conflict and fake conflicts. Severe conflicts are real conflicts, while fake conflicts are when the read-set intersects with the committed write-set but is in fact not a collision. An example is the serializable history H_2: $T_1r(a)$, $T_2w(a)$, $T_1r(b)$, shown in chapter 5.2. The read set to T_1 is {a, b} which intersects with {b} (the write set to T_2). Many protocol reports this as a conflict because of weaknesses. EOCC reduces the number of restarts because of less fake restarts.

Illustration 5.5 shows an example of how EOCC avoids fake restarts with transaction T_1: w(x), w(c) and T_2: r(c), r(a). First T_1 commits, then T_2 starts. In the upper example (the old approach) is the new value to item c not broadcasted before next cycle. Therefore the old value is read, which causes a restart when next CI is broadcasted. This restart is fake because the value a is still consistent with the old value of c.

The lower part of the illustration and the new approach, the new values to a committed transaction are ready for broadcast right away. Therefore the new value to c is read, next a is read and T_2 commits successfully. The fake restart is avoided.

If T_1 committed after T_2 read c, then c would obviously be the old value. The new value given by T_1 would not be broadcasted before next cycle, but then T_2 would already be finished except for the last validation. The validation for T_2 at the last CI would announce that the value for c in the previous broadcast was new. As far a we understand, this last validation will lead to a restart, because the read set in T_2 intersects with the write sets in the last CI. So there are still some room for improving and avoiding more fake restarts. We

discuss this optimisation in the last section of this chapter.

The broadcast cycle is divided into many sub periods (SP). Between each SP a fixed space is reserved for the CI. The previously committed values since last CI are broadcasted in the CI. If it is filled up over the reserved space, then a bit is used to indicate that the rest of the CI is put in the next reserved space. The committed write set is included in the broadcast cycle immediately. This technique is discussed in the last section of this chapter.

The reducing of fake conflicts and the partition of the broadcast cycle into sub periods leads to good performance. The simulations in [53] shows comparison between EOCC, FBOCC and conventional OCC, and EOCC has the best performance.

The backside of the protocol is the fixed reserved size for the CI. In times with few updates, the CI will be nearly empty, and therefore waste space. In times of many updates, the CI will be overloaded. In a stable environment with a steady rate of transactions, the size of the CI can be chosen in such a way that little space is wasted, but in environments with bursts of transactions, the size must be chosen big enough to avoid CI info to be delayed too long (for example one broadcast cycle). The selection of the number of sub periods and CI size is marked as future research in the scientific paper by Guohui et al.

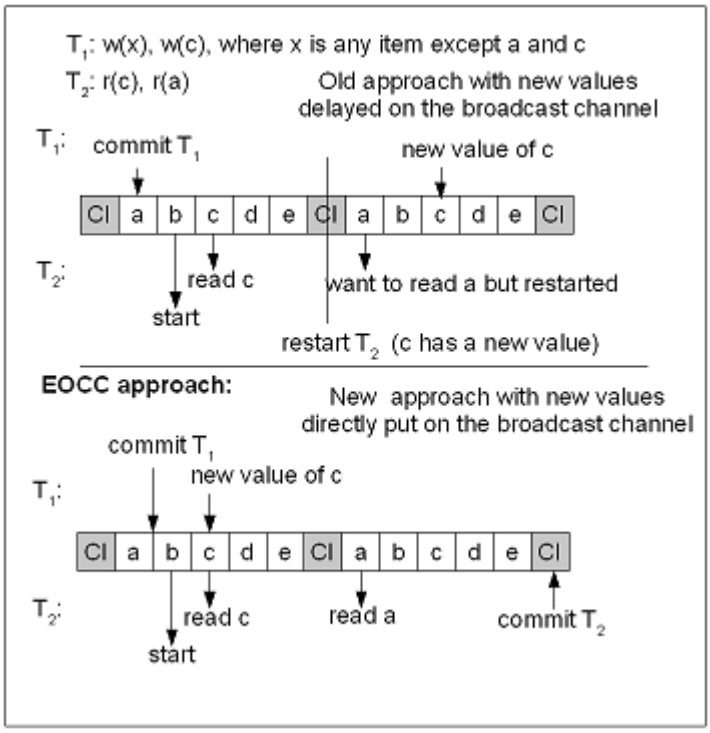


Illustration 5.5: Example of how EOCC avoids fake conflicts

5.15 Concurrency Control Protocol Summary

A summary of the characteristics to the reviewed protocols is shown in Table 5.12.

	Real-time	Not real-time
Serializable:	EOCC, FBOCC, PVTO Read-only: BCC-TI	Certification Reports Read-only: UFO, Multiversion
Mutual consistency and currency:		F-Matrix, R-Matrix, (OCC-TI)
Local and Single serializability:		STUBcast

Table 5.12: All protocols supports update transactions, unless "read-only" is written

In order to get better performance, more transactions should be committed, and less rejected. The approaches we have seen to accomplish this is to :

- Relax the correctness criteria
- Have better evaluation algorithms to guarantee the correctness criteria will:
    - Avoid unnecessary restarts
- Detect conflicts early in order to avoid late transaction restart [44]. Wasting of upload bandwidth is also avoided when the conflict is detected on the client side.

Other techniques are summarized and discussed throughout the rest of this chapter.

5.15.1 The CI and its Contents

The review showed us it is efficient and scalable to let the clients process and commit the read-only transactions locally. This can be done by transferring CI (control information) to the clients in the broadcast. The CI contains information about the recently committed transactions, such as write set. The amount of

data in the CI varies from protocol to protocol, but in general a small CI is desirable so the broadcast cycle is smaller.

The CI is essential for early restart detections, and helps the clients to validate its own transactions and commit them locally. Client autonomously and committing locally is very scalable.

5.15.2 The Placement of the CI

Two techniques are possible when it comes to the placement of the CI. The first option is to broadcast it *before* the recent committed updates, such as FBOCC, PVTO, APPROX (F- and R-matrix), Datacycle and BCC-TI, and announce the data items that are new in the next broadcast cycle. The second option is to broadcast the CI *after* the recent committed updates, such as EOCC and STUBcast, and announce the data items that were new in previous broadcast cycle.

The first approach has the advantage that once the broadcasting of one cycle starts, the data set is consistent, and new updates are not broadcasted in this cycle. With other words, the data set in one broadcast cycle is always consistent, and by looking at the CI, the consistency when reading the next broadcast cycle can be determined. The backside is the late propagation of new values, because they are in a worst case delayed almost two whole broadcast cycles (see Illustration 5.6).

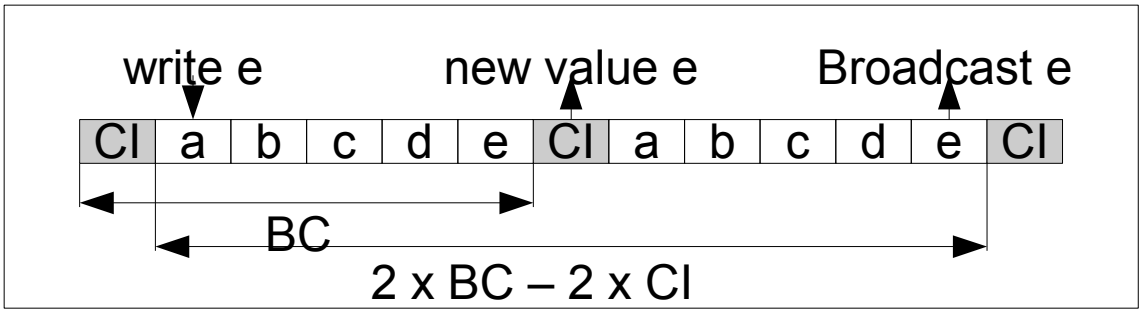


Illustration 5.6: Extreme case of delayed broadcast of new value when CI is placed before the new values

The second approach has the advantage of propagating new values straight away. If the new values are broadcasted in a special kind of CI (or ucast as STUBcast names it), then the client must listen constantly to the channel because it does not now when the update arrives. If the new value is simply put out on the channel, and not announced as a new value before the CI, then the clients must always wait for the CI to check for consistency before committing. That again will lead to a burst of transactions at the server after each CI. When the service gets swamped some clients have to wait for others to be finished with their transaction commit.

By broadcasting many small CIs in one cycle (like EOCC does), the burst will disappear as well as the clients must not wait so long before it can validate the results. The use of many sub periods and CIs is covered separately in the next part.

5.15.3 Partition of Broadcast Cycle Into Sub Periods and Several Sub CIs

The newest protocol (EOCC) is the only one which has many small CIs distributed equally on the broadcast cycle. Each CI has reserved space for the write set to recently committed transactions. The simulation results they presented showed a performance increase. The clients do not have to wait so long for the last CI and validation. The transactions are therefore committed faster and conflicts are detected earlier. The burst effect is also eliminated since the CIs are distributed equally over the broadcast cycle.

The number of sub periods and size of CIs have a big impact on performance and must therefore be chosen carefully. With many sub periods the clients must tune in more times during an active transaction but for a shorter time.

If the time it takes to tune in to the channel is big, then the CIs shoulc not come too close. The 802.11g has an overhead at 20 µs to tune in where as 802.11b has an overhead of maximum 192 µsec [54]. In addition, higher layer header (20 bits for IPv4 and 40 bits for IPv6, plus 4 bit checksum for both IP versions) must be considered. If the database has 1000 items and is broadcasted in 1 second, then one item is broadcasted in 1 msec. So the time it takes to tune in to the broadcast channel is almost neglectable for 802.11g.

When the broadcast cycle is divided into four sub periods, then the clients have to wait four times shorter in average. To shorten the time four times more, the sub periods must be expanded to 16 (a quadratic

increase). The number of sub periods should therefore not be exaggerated, since the decrease in time is much less then the increase of times the clients must tune in to the channel.

5.15.4 Real-time and Server Update Transactions

Another feature worth to notice is the forward validation at the server side in FBOCC. It is a nice way to integrate prioritizing of restarts (real-time) and to treat the server transactions because they can be restarted early.

5.15.5 Partial Restart

When a transaction discovers a conflict, it means one or more of its operations are invalid. The normal approach is to restart the whole transaction, and start it from the beginning again. We suggest to only restart a part of the transaction. The effect of executing fewer operations in a partial restart in comparison to a total restart, is a lower transaction time (total time to complete a transaction, see 6.4 Simulation Settings for full definition). Less operations cause the battery powered devices to use less electricity, and lower transaction time leads to bigger probability for more real-time transactions to meet their deadlines.

Because the operations in a transaction is decided out from the value of the preceding operations, it is not enough to restart only the conflicting operation. The value of the conflicting operation has changed, so the following operations are probably no longer done on the correct data items.

But the operations before the conflicting transaction are still the same, and as long as they are valid it is not necessary to read them again. We define *partial restart* as the following approach:

Instead of restarting the whole transaction when a conflict is discovered, it is only necessary to restart from the operation that caused the conflict. If several conflicts arise, the transaction should restart from the oldest operation (earliest in the transaction order) that causes a conflict.

This approach will always maintain the serializability as long as the non-restarted operations have valid values. That is self-evident. The challenge of assuring the correctness to the non-restarted operations is dependent of the protocol the approach should be implemented in. Several of the reviewed protocols can implement partial restart fairly easy. We show an implementation into FBOCC in 6.5.5 Partial Restart in FBOCC. There the CI is used to guarantee the correctness of the non-restarted operations.

Techniques similar to this approach are *rollback recovery* and *checkpoint scheme*.

Rollback recovery is an old technique used in databases to rollback operations done by a failing process and Fussel et al. [55] used rollback to remove deadlocks in lock-based database systems, instead of restarting transactions.

Checkpoints are used to reduce the loss of computation in the case of failure in a distributed system [56]. The checkpoints are a point which all the processes can fall back to in case of a failure, but in contrast to *partial restart* the checkpoints is a global state of messages in a distributed system (for instance mobile clients). The partial restart applies to a database set that is shared by many users. The values can change at any time, so it is hard to guarantee serializability.

5.15.6 EOCC Fake Restart Improvement

Here we continue the discussion of fake restart avoidance in 5.14 EOCC from the last case where T_1 committed after T_2 read c. Since T_2 only read item ***a*** as a last operation and do not have any operations on c this last cycle, it could have committed successfully. The reason for the restart is in case another value y are read later, a dependent conflict could arise between c and y.

For example if another transaction T_3 did r(x) and w(y), and T_2 did a r(y) afterwords, the conflict of y would not be discovered. The conflict graph would be T_2r(c) -> T_1w(c) -> T_1w(x) -> T_3r(x) -> T_3w(y) -> T_2r(y), or simplified as T_2 -> T_1 -> T_3 -> T_2 which is clearly cyclic.

So if r(a) was not the last operation, then a restart should be done. If r(a) was the last operation, the transaction could commit.

5.15.7 Server Validation Answer on Dedicated Back Channel or Broadcast Channel

PVTO and UFO assumes there is no dedicated back channel (from server) to the clients, so the answer of the validation is sent on the broadcast channel. This is of course the only alternative in case there is no back channel, but when such a channel is available, is it more efficient to use it? We have not conducted any tests on this area, but by sending the result directly, the client would get the answer faster. Also, the broadcast cycle would be slightly smaller because no space for accept and reject messages needs to be reserved.

6 Simulation Platform and Simulations

This chapter describes the structure of the simulation model. The simulation engine (CSIM) that were used to make the simulation model is first introduced. Then the structure of the simulation model is described, part for part, explaining how the various components are implemented and defending the correctness of it. The description of how the testbed is set up with multiple runs and new random seed is last in this section.

6.1 CSIM

The simulation model is built by using the simulation engine CSIM18 from Mesquite Software [57]. CSIM is a commercial product written in C, and is a library of functions, procedures and header files. The choice of simulation tool fell on CSIM18 because it was the only one available to us from the university. It is a much used tool in this research environment and no free alternatives exist. CSIM18 is available for a multiple number of platforms, amongst them both Microsoft Windows and Linux. We were given the Windows version and used *Microsoft Visual Studio 6.0* as development environment.

CSIM version 18 provides simulation components such as processes, mailboxes, events, random numbers and tables. An in-depth description of these components is found can be found in the CSIM documentation [58].

- A process is an independent thread of execution and in simulation the processes appear as running in parallel.
- Mailboxes are used to interchange information between processes. A mailbox has an identifier and can only transfer a *long* value. In order to transfer bigger amounts of data, the *long* value is used as a *pointer* to a structure. All processes can read and write to all mailboxes. The messages will be queued up until they are read. When a message is read, it is also removed from the queue.
- Events are used to synchronize actions of different processes. Processes can wait for an even to happen, and when this happens the process will continue running again and the event will be cleared.
- Random numbers can be generated for 18 different distributions families (such as uniform, poisson, or exponential) and by changing the parameter values most distributions are available. For each execution of the simulation model, the same random number is chosen. This makes it easy to reproduce a specific run or error. In order to get new random values (which is indeed necessary to get statistical material for calculation of average values), a new seed can easily be inserted.
- Tables are used to collect data values and to report on their statistical properties. A standard report can be generated and histograms and confidence intervals can be requested. Normal procedure in the research environment is to use the numbers generated by the tables to generate a graph in a third-party program (such as an advanced spreadsheet application).

Time is simulated with time ticks and is defined as a double value. The time stands still when a process is executing and advances when a process is not active. To be exact, the time advances when a process executes a *hold(double ticks)* statement, or *receive(MAILBOX mbox)* when no messages are in the queue and the timeout is more than zero if it is present. This is explained in more detail in the following text.

A process is exclusively executed until it executes a hold statement or a receive statement which receives from a mailbox without any messages. When a message is sent into the mailbox, the receiving process will get the thread of execution and be active within the same time tick. One exception is by the use of timeout. Then the receiving process will get the focus back after it has waited the time specified by the timeout. If the timeout is zero, the process will check for a message, and continue right away without ever being task switched.

6.1.1 CSIM Limitations

Some technical issues must be remembered when programming in CSIM. For example is the following taken from the CSIM documentation [58]

“The process manager preserves the correct context for each instance of every process. In particular, separate versions of all local variables (variables resident in the runtime stack frame) and input arguments for a process are maintained. CSIM accomplishes this by saving and restoring process contexts (segments of the runtime stack) as processes suspend themselves and as processes are "resumed" (restored). A consequence of this kind of operation is that if one processes passes an address of a local variable to another process, it is likely that when this address is referenced, the reference will be invalid. The reason is

that when a process is not actually computing (using the real CPU), its stack frame with the local variables will not be physically located in the correct place in memory. This is not a major obstacle to writing efficient and useful models; it is a detail which must be remembered as CSIM models are developed."

To go around this limitation, some variables are declared global instead of local.

CSIM documentation also states the following;

"A message can be either a single long integer or a pointer to some other data object. If a process sends a pointer, it is the responsibility of that process to maintain the integrity of the referenced data until it is received and processed."

Which means it is possible to send a pointer to a local variable, as long as that process maintains its integrity.

In the beginning we discovered wrong execution of the CSIM model when using compiler optimization. We reckon it might have something to do about how the context switching is done to simulate multiple processes.

Later we received some memory access violation in the CSIM code during execution. Because CSIM is a commercial simulation tool, we only have access to the precompiled library, and are therefore unable to debug error that occurs inside the library code. In our case it turned out that accidentally a statement overwrote a piece of the memory area used by CSIM variables, which in turn caused the CSIM to execute falsely. But if errors exist in the CSIM code, a big limitation is that it is impossible to fix without the source code.

6.2 The Platform Structure

The simulation model consists of one server with one database, one or more clients, and a structure to simulate the broadcast channel.

The structure of processes and mailboxes are shown in Illustration 6.1. Each process is drawn as a rectangle and a mailbox as an arrow. It is easy to see that for each new client, a total of 3 new mailboxes and 2 new processes is needed. For each new process one new event is possible.

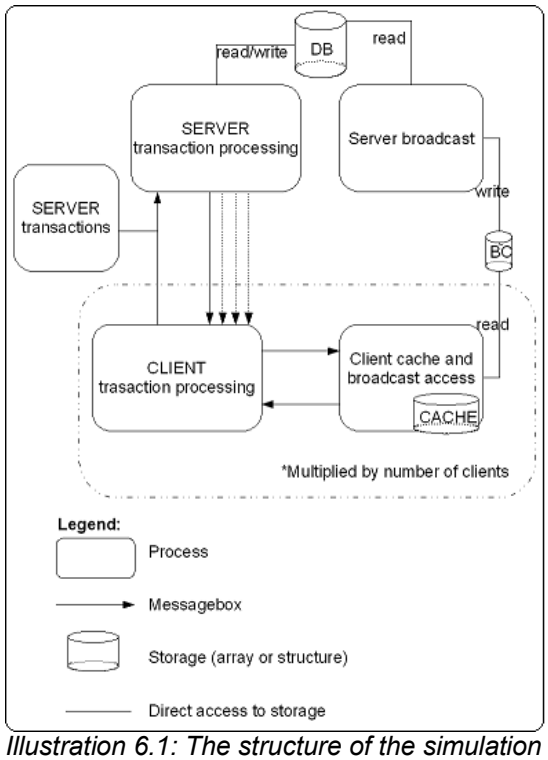


Illustration 6.1: The structure of the simulation model

6.2.1 The Server Structure

The server consists of two processes, one main process for transaction processing and one process for the broadcast control. The broadcasting process only puts correct data on the broadcast channel, so therefore it works independently of the rest of the server and needs no communication except to read next broadcast data from the database.

The two server processes access the same database which contains all the data items. The broadcast process takes this data and makes an broadcast item out of it. The simplest way to do this is just broadcasting the data in a cyclic manner, but it is also possible to divide it into multiple broadcast disks as proposed in [5] and [6]. The database is modelled as a global array to make it accessible by all the server processes and still guarantee consistency when CSIM copies the memory areas to simulate multiple processes running simultaneously.

In some protocols (e.g. FBOCC [10]) the clients requires the transaction validation answer directly from the server, and not through the broadcast channel. The server can send messages to the clients through message boxes, one dedicated for each client.

To deal with server update transactions, a new process was made. The process has the task of creating and processing server transactions. These transactions are then again validated against the running client transactions and if collisions occur actions are taken according to what the protocol describes. The server transactions gets validated against all the running client transactions, as well as the database items.

6.2.2 The Client Structure

The client consists of two processes, one main process for transaction processing and one to process cache and broadcast access. The two client processes communicates through two message boxes, one for each way. A transaction starts at the client main process and requests for values are sent to the *cache and broadcast* process. The mailbox *cacheMailbox* is used for this communication. In order to keep the same structure on all messages independent of the protocol in use, a pointer to a structure called *sinternClient* is sent as payload of the message each time. The structure consists of a char to define type of message and a new pointer to the real message. The message corresponds to the type which is defined in type. This way a more general model is achieved and it is easier to implement other protocols. This strategy is used several places.

The clients send transactions to the server through a mailbox that is common for all the clients. Since we are working in a virtual time created by CSIM it is only needed to have a single threaded server. A multi threaded server can be simulated by reducing the transactions validation time.

Several other simulations models made for concurrency protocols, in e.g. [7], use only one client. They defend this choice by saying several clients can be simulated by reducing the waiting time between each transaction. This assumption works only for some read-only environments or if one client can execute several transactions concurrently and without interaction between the active transactions on the client (because an assumption in the broadcast environment is that there are no communication or synchronisation between the clients). PVTO [45] seems to use one client which can execute several transactions concurrently and independently. We need to simulate conflicts between the client transactions. To do this we create multiple processes of the client type, this is both quite easy to do and it will provide a real life situation, but at the cost of a longer simulation run time.

6.2.3 The Broadcast Channel

The broadcast channel is modelled as a global structure. An element which is to be broadcasted is put in the broadcast structure by the *server broadcast* process. The clients then reads from the structure at one specific time defined as the broadcast time. This is right after the server has broadcasted. The data is read in one instant, but the client must still wait one *broadcast time* before reading next element (in order to simulate the time it takes to read the value on the fly from the air.). Therefore the result will be as close to a real environment as possible.

The broadcast items are read from the item database. The server must make sure that the data that is copied out is consistent. This problem is taken care by CSIM because a process is exclusive executed until otherwise specified with a hold or receive statement. That means a transaction is fully executed at the server process before another process takes over.

6.2.4 Caching

Each request for a database element is done through the client cache process. If the element is in the cache, the element is returned instantly. Else, the element must be found at the broadcast channel. An element found from the broadcast channel can be put into the cache database, depending on the cache algorithm. The caching database is modelled as a local array.

The mailbox from the main client process is checked for each time tick at the cache client process. The check has a zero time tick timeout something that means it will just check the message queue and take the first message there if there are any waiting, else wise it will just continue the program progress immediately. This to make sure that the broadcasted items gets stored in the cache if the cache algorithm specifies it.

6.2.5 Concurrency Protocols

The concurrency protocols must be implemented at both the client and server side. The main validation is usually at the server side (main server process), and pre- or forward validation is at the client side (main client process). The pre validation is done on basis of control information which is broadcasted by the server. So also the broadcast data needs to be adjusted. The validation answer is sometimes given in the broadcast data (e.g. PVTO [45]) and sometimes directly through a dedicated channel (e.g. FBOCC [10]).

6.3 Protocol Implementations

All the protocols have special features which requires some special programming. They all have different structures of the messages transferred between the processes and the broadcast. Therefore we were unable to present a basis framework ready for implementation of a new protocol. Also various data management protocols must be very closely integrated with the concurrency protocols. So each implementation must adjust and tweak variables and structures all over the model. For future

implementations we offer the implementation of the current state of the art protocols we chose to implement. They can be used as a base to implement new protocols, and can be used as reference and comparison foundation.

In the specific implementation of a protocol it is usually only to follow the pseudo code provided in the description of the protocol. Sometimes it is convenient to do some shortcuts to simplify the implementation. They are in that case described in the comments.

In this section we describe an overview of the implementation and special considerations of the protocols we chose to implement. We will also write a few words about our experience of implementing the protocol. For a more technical and specific information about the implementations, we refer to the source code and its comments. It is assumed the reader have read about the protocols in chapter 4 and 5.

All protocols are processed until they are committed. If they receives an abort, the same transaction is restarted.

6.3.1 Data Management

As a basis, broadcast disks were implemented. Since it can be theoretically derived that binary-based broadcast performs better, or equally good in all cases. And we know that they perform equally good when broadcast disks have no empty slots. So by choosing the right values for the disk frequency it is known that we get the same result with both methods. It is easier to implement broadcast disk, and we can just conclude that the performance of the broadcast can be increased by using binary-based broadcast instead.

The indexing methods (1, m), tune optimal and latency optimal were implemented. And small tests were conducted to test the validity and performance of these to confirm that the results correspond to the expected theoretical values.

Latency optimal, tuning optimal and (1, m) indexing were also implemented. This to test how they would impact the performance gain gotten from using broadcast disks compared to a normal cyclic broadcast.

But there were no groundbreaking results when we simulated these data management methods, and there are quite a few good comparisons published already. So the simulations in the report focus on the concurrency control protocols, but the simulations is still a part of the source code package so the simulations might be run if needed.

6.3.2 Broadcast Disks

We implemented a broadcast disks with two active disks. The frequency and the size of the disks can be changed easily. We conducted some tests together with the concurrency protocols. The tests are described in 6.3.9 Broadcast Disk with Concurrency Protocols.

6.3.3 Concurrency Control Protocols

We decided to implement BCC-TI, FBOCC, PVTO and STUBcast. There were tested against each other with different environment parameters.

We chose FBOCC because it is state-of-the-art and one of the best performing protocols. EOCC performs better according to [53], but in order to implement EOCC we would first have to implement FBOCC and make an expansion to it. If we had more time, we should have implemented EOCC too.

PVTO was chosen since the performance looked like it was as good as FBOCC and sometimes even better when looking at the simulations results presented in [45].

STUBcast is a relatively new protocol and it is not compared with other protocols (as far as our knowledge goes). And most of all, STUBcast uses a relaxed correctness criteria in order to increase performance. Many scientific papers have discussed if it is useful to relax the correctness criteria or not, and if it has any performance increase at all. We chose to implement and compare STUBcast, in the search of the answer to these questions.

BCC-TI is a read-only protocol, and we wanted to see how it performed in comparison to more general protocols. By general protocols we mean those who apply to more or all defined environments in chapter 5. Examples are FBOCC and PVTO, which support *client update transactions*, *real-time*, and *global serializabliity*.

All of the protocols in the following chapters are implemented with normal cyclic broadcast. The full source code for all the implementations are distributed together with this report.

6.3.4 FBOCC

This was the first protocol we implemented, and therefore also the most time consuming part. This is also what we used as a basis for our further implementations. It has both forward and backward validations. Forward validations with the ongoing server transactions and backward validation against the previous transactions. To make the backward validation possible it is needed to have an memory that stores the old transactions and the size of this memory is dependent on how long a client needs to be able to complete an transaction. One possibility is to say that if the transaction is to old to have a backlog to compare it against it should just be rejected. But this could seriously hurt environments where there is a great overload on the server. Our approach is to just have the size as a static size and just print out a message if we get any transactions that are older than the backlog. This makes it possible to just ignore the fact that we only have a part of the backlog for the item. If the backlog is of a decent size this should not provide any problem at all, since the transaction would probably be rejected anyway. And it would only be a very small number of the measurements that are affected by this so the results would still be valid.

To simplify the implementation of the server transactions we used globals for communication between the *server* process and the *serverTransaction* process. This simplifies the process and saves us from creating an extra thread. And the *serverTransaction* process also have direct access to the database and control information so everything is just manipulated directly without any need to go trough the *server* process.

Since FBOCC uses control information we needed to add an additional broadcast message. This message is sent in the start of each broadcast cycle and describes what items that have been invalidated during the last broadcast cycle. The client then checks its ongoing transaction against the control information to see if there are any conflict. If there are conflicts the transaction is restarted.

6.3.5 Partial Restart in FBOCC

We have implemented *partial restart* in FBOCC to measure the performance increase and to show an example of how to implement it. FBOCC can easily implement partial restart because it is know which item that cause the conflict and the CI provide information enough to decide the correctness of the non-restarting operations.

The *partial restart* can be done both at the client and the server. The server must reply with a rejection message including the conflicting operation id. We have only implemented *partial restart* on the client side because this technique was discovered late in the project period and time was limited. Write operations are first validated at the server side, so we will only look at the read operations on the client side. (In case blind write is not allowed, a write operation will always be preceded by a read operation. Then the read operation will implicitly evaluate for the write operation). The implementation on the client side in FBOCC is enough to illustrate how to implement the technique as well as indicating the performance increase.

FBOCC sends a CI in the beginning of each broadcast cycle with the write set of committed transactions in previous cycle. A client restarts when it finds an intersection between the local read set and the write set from the CI. The intersecting read operation is the one that causes the conflict. In case of several conflicts, the oldest operation must be chosen as the conflicting one.

So instead of restarting the whole transaction as normally, the operations before the oldest conflicting operation are kept, and the oldest conflicting operation is read again (restarted).

In order to guarantee the correctness of the non restarting transactions, the client must listen to all the CI and do the validation.

An example of a transaction T_2 with a conflict is shown in Illustration 6.2. Two alternative restarts are given, normal and partial. T_2 does the following operations; r(b), r(a), r(d), r(c), r(e), and is restarted before the last operation r(x) can be done. Another transaction commits w(d) and w(e) in the third broadcast cycle, which causes a conflict for T_2. The CI write set intersects with T_2 read set, $T_2[RS] \cap CI[WS] \neq \{\}$, which indicates a conflict, so T_2 is restarted.

The partial restart is from the oldest conflicting operation. In this case both r(d) and r(e) caused conflicts, so the transaction is rolled back to r(d) because it is first in the history of the two, and hence the oldest. The new value to item *d* is read, and the next operation is determined out from the value of *d*. In this case, the next operation turned out to be *e*. The value to *b* and *a* is not re-read because it is the same as long as no conflict is discovered in the CI information. If a conflict arises in the next CI for let say item *b*, then the partial transaction will roll back and restart from r(b), which in this case is the same as a normal restart.

In Illustration 6.2, the normal restart wastes resources and time by reading the same values for item *b* and *a* again.

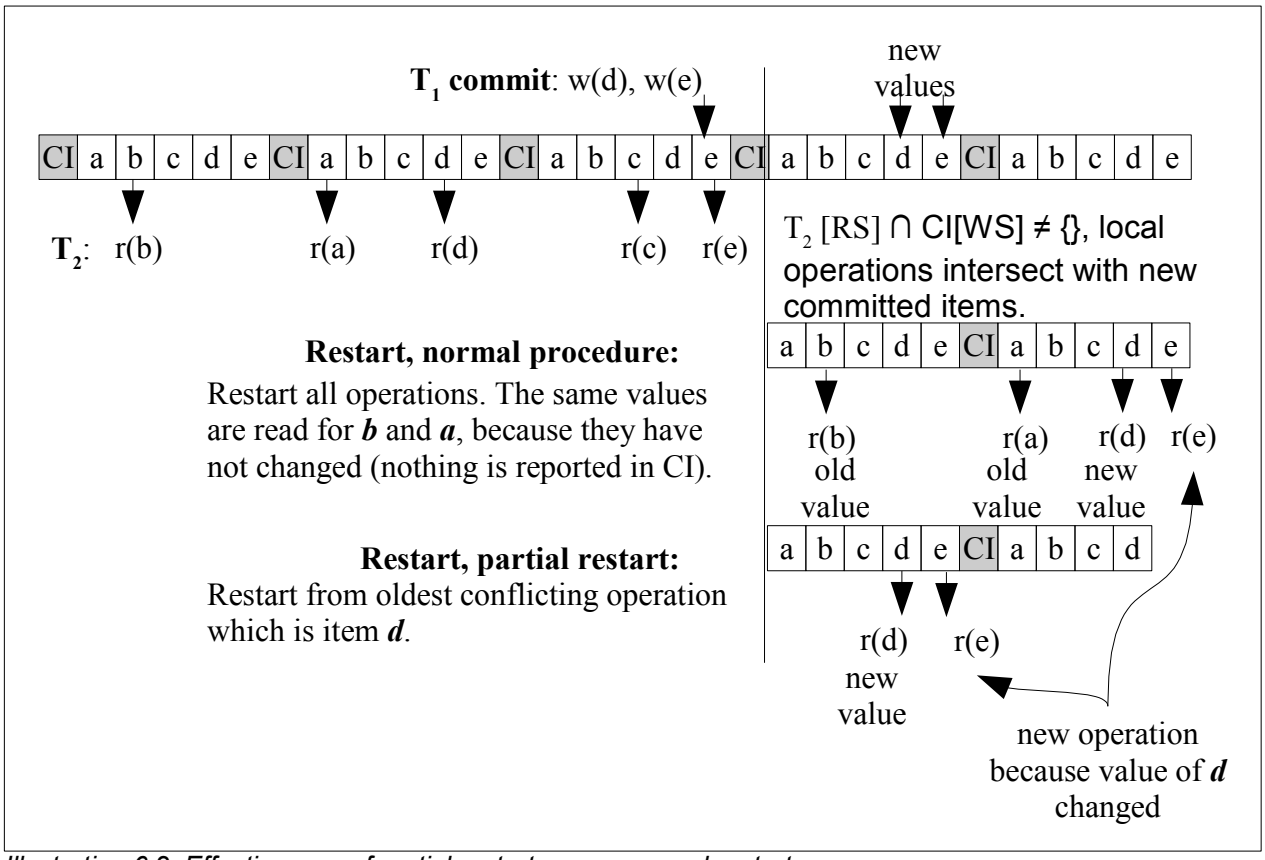


Illustration 6.2: Effectiveness of partial restart versus normal restart

6.3.6 STUBcast

By cutting down on the serialization requirements STUBcast tries to get an better performance in environments where less restrictive serialization requirements are possible. We wanted to implement this method since it is not compared against other methods. The environment the authors used to test it is totally different from how other methods are tested, so we implement it and test it against other methods in similar environments. By doing this we hope to be able to test if this cut down on serializability can help on performance.

STUBcast is implemented in the standard framework we made, and some minor adjustments needed to be made for it to work. First of all we needed to make support for the data propagation STUBcast uses. The broadcast process needed to receive the data from the server process and send it out to the clients as soon as possible. This required two additional message types on the broadcast. One for the data propagation and one for the UBE (message that describes the reads in the transaction and also signals that the data propagation is finished). The client receives these and stores the timestamp in a array. So at all times the client have an array with the timestamp for all the data items. There were also need for quite a few extra arrays to store information, but the model were never made with memory conservation as a guideline so in many places we use indexed arrays to save simulation time. But the method still uses quite a lot more memory on the client side than most others.

Since STUBcast does not define how server transactions are to be implemented we just have to reckon that they are absent or performed in a way that does not affect the client transactions at all.

6.3.7 PVTO

The characteristics of PVTO is the high amount of data values, (such as validation set, TOR and timestamps), which must be maintained and transferred. The validation process is quite complex too. The client validation is done at the *cache and broadcast* process, because the validation must be done continuously for each control information. The operations are initiated by the client main process and a request for a data value is sent to the *cache and broadcast* process. For each request, the reply can be either the value or a *SIGNAL_ABORT*. At the end is the data values (validation set, TOR, etc.) sent back to the *client main* process.

The server validates and broadcast data and control information. The accept and reject set in the control information only contains the client number because only one transaction can be active at one time at one client.

6.3.8 BCC-TI

BCC-TI is a read-only protocol, and have relatively easy validation. All validation is done autonomously on the client. The validation is done in the same way as for PVTO, that is on the *cache and broadcast* process.

The server do only accept transactions from the *serverTrans* (Server Transaction) process and propagate new values to the broadcast channel. Each server transaction is assumed to be committed with a lock based protocol. We implement this by letting only one transaction to be active at a time, and the duration of one server transaction non-existent. It is executed and committed atomically. The important thing is that the database set gets new values, such that conflicts arise at the clients. This is accomplished.

6.3.9 Broadcast Disk with Concurrency Protocols

We implemented broadcast disk with BCC-TI, FBOCC, FBOCCpr and PVTO. We also had to change to model to include support for a none uniform access pattern for broadcast disks to have any effect. We implemented two disks, one slow and one fast. The item distribution among these can be easily adjusted and the same with the disk frequency that controls how fast the disks spin compared to each other.

6.4 Simulation Settings

The simulations are performed using the environment variables described in Table 6.1 and we compare how the different methods perform when we change the transaction length. It is important to notice that some protocols will in many cases get very good results compared to the other methods because of more loose restrictions in form of serializability, server transactions etc.

We measure four different results:

- **Item wait time**

  The time it takes to get an item from the broadcast when it is requested. This value is only affected by the size of the broadcast and will therefore only vary if the broadcast size is increased because other messages than just the data are broadcasted.

- **Transaction time**

  The time it takes to get a transaction accepted. This time also includes the time spent on restarts of the transaction. Both client side restarts and server side restarts.

- **Transaction throughput**

  This describes how many transactions that gets accepted per time unit.

- **Number of restarts**

  We also measure the restarts. Two kinds of restarts are measured, first we have the restarts made at the client side. These restarts will increase the waiting time for a client, but will not affect the load on the server in any way. We also measure the number of restarts that occurs when the server validates the transactions. This kind of restarts means that the server eventually will have to do another validation and therefore this kind of restarts will result in both increased load on the server and increased waiting time for the client.

We look at how the different protocols perform compared to each other when the transaction length increases. This performance is measured in number of restarts, where a low number is the better one.

The ratio between read and update transactions is set to 1:1 so half of the transactions are purely read only. While this is a very high ratio of update transactions compared to a real life environment. We argue with the fact that read only transactions do not put any additional load on the server in most of the protocols and we can therefore have a high update ratio to avoid having to use so many clients to be able to get good simulation results.

We use a flat scheduling for all the tests since we have no real time considerations so there is no way to decide what items is the most critical to broadcast. And we want to measure the performance of the concurrency protocols so using an advanced scheduler would just introduce another source for errors.

The access distribution in our tests is uniform so all the items are equally hot. This will typically result in better results than an none uniform distribution would give, but this is something that affects all the

protocols in more or less the same way. To improve performance in a none uniform environments we could use broadcast disks instead of the simple cyclic broadcast pattern as we describe in chapter 6.3.9 Broadcast Disk with Concurrency Protocols.

6.4.1 Environment

Name of environment variable	*value*	*Description*
Environment variables		
NUMBER_OF_CLIENTS	100	The total number of clients that should be started
TIME_PERIOD	5000000	How many time ticks the simulation will run
Client variables		
READ_ONLY_TRANSACTIONS	0.5	Percentage of the the transactions that should be read only
MEAN_INTER_OPERATION_DELAY	20	Delay between each operation in a transaction
MEAN_INTER_TRANSACTION_DELAY	200	Delay between each transaction at a client
TRANSACTION_LENGTH	6	Transaction length
Server variables		
BROADCAST_TIME	20	The total time it takes after a broadcast is sent out until the next broadcast is sent out
DATA_BASE_SIZE	300	Total number of data items in the broadcasted database
SERVER_TRANSACTION_LENGTH	8	Length of the server transactions
SERVER_MEAN_INTER_OPERATION_DELAY	1	Delay between each operation in a transaction on the server transactions
SERVER_MEAN_INTER_TRANSACTION_DELAY	1000	Delay between each sever transaction
TRANSACTION_VALIDATION_TIME	10	The time it takes to validate an transaction

Table 6.1: Environment variables

All the common environment variables are declared in a separate header file named “platform.h”. The environment settings for one protocol can easily be changed in this file. If a protocol requires some additional environment variables then these will be declared at the top of the C file.

Table 6.1 lists the environments variables and a short description of their purpose.

We study the effect on an increased number of clients and how that will affect the performance. And we also use different length on the client transactions to see how this will affect performance to.

We have not implemented any real-time deadline on the transactions. The number of restarts will indicate how many transactions will meet their deadline. If the transactions miss their deadline or not is highly dependent on the ability to prioritise critical transaction in front of others.

6.4.2 Simulations

We run the simulation for 5,000,000 time ticks for each sample and with 100 clients. That gives us between 120,000 and 170,000 accesses to the broadcasted data, dependent on how much other information that is broadcasted. And we also get between 5,000 and 80,000 transactions, this will of course vary a lot dependent of the transaction length and what kind of method that is used. This big sample space gives a very good confidence interval and eliminates errors that occur because of badly generated random data. We can also argue that we perform sufficient runs to get good measurement values since the model itself provides a bigger source for errors.

To ensure the correctness of our tests we did 10 runs of the same test on BCC-TI, each with a different random seed so that the test were run with different random data. Table 6.2 Shows us the different results from these tests, from these we can conclude that all the different results fall into an area that is 0.5% of the average from all the runs and can therefore conclude that a single run will provide good enough results alone. By comparing the results against the other tests of the same protocol and looking at trends we can also identify potential errors caused by erroneous input parameters.

Test number	*Average transaction time*	*Average transaction time for transactions without restarts*	*Percentage of client transactions accepted*
1	12656,99	11601,66	95,09
2	12688,90	11636,64	95,15
3	12657,04	11586,90	95,06
4	12662,00	11647,11	95,34
5	12644,93	11629,64	95,31
6	12694,20	11603,54	95,03
7	12667,92	11607,47	95,11
8	12692,27	11612,14	95,05
9	12685,18	11636,15	95,20
10	12666,01	11639,27	95,30
Variance	49,27	60,21	0,31
Percentage variance from the average	0,38%	0,51%	0,33%

Table 6.2: Variance in test results

A small test were performed to test the effect of warm up time (the time it takes for the simulation to stabilize) in the simulations to make sure our results would be valid. This could be an important factor., specially with very long transactions. But it seemed like the results were not affected by the warm up time at all, there were less variance in the results when we removed the warm up time than there were with a different random seed. We also used the built-in functions in CSIM to calculate confidence intervals for the results, this to provide a certainty that the results were correct. If the results had a to big confidence interval we either rejected the results, or we had to do a longer simulation.

The results from the tests were collected and compared against each other to make sure that no erroneous results occur in them. The results from the tests were then again plotted in a graph using *OpenOffice Calc [59]*.

6.5 Evaluation, Tests and Results

In the following sections we have the results of the various tests and discussion over the results. We adjust the transaction length, finds a performance measurement for our framework, adjust the number of clients, adjust the database size, and test out none uniform client access on the database (multiple broadcast disks).

BCC-TI is a read-only protocol, and therefore needs server transactions to get the database set updated. STUBcast do not have server transactions in the model, so BCC-TI and STUBcast can not be compared directly. We suggest one alternative approach to overcome this problem in 6.5.2 Transaction Length for Qualitative Characterisation Framework.

All simulations are done with a standard environment values(given in 6.4.1 Environment) if nothing else is stated. All transactions are executed until they are committed. So for each abort, the transaction is restarted.

Because of the huge average transaction time on some of the methods we were not able to get a confidence interval for them, but the samples will still be shown to give a general view on what values to expect.

6.5.1 Transaction Length

One of the environment variables that affect the performance the most is the transaction length. An increase in the transaction length will always lead to a great increase transaction time and when the environments has updates, the abort rate will increase as well.

For high values of the transaction length, very few transactions are committed, so the confidence interval is not satisfactorily. We show the values anyway, because they do indicate the direction. For transaction length 8 the confidence interval is 90%, and as the transaction length increase, the confidence interval decrease.

6.5.1.1 Partial Restart

Illustration 6.3 shows that the longer transaction there is, the better the improvement is with partial restart. This is also logic, because the longer the transaction, the greater chance of getting a conflict. And as long as the conflict do not appear on the first operation partial restarts will save some time.

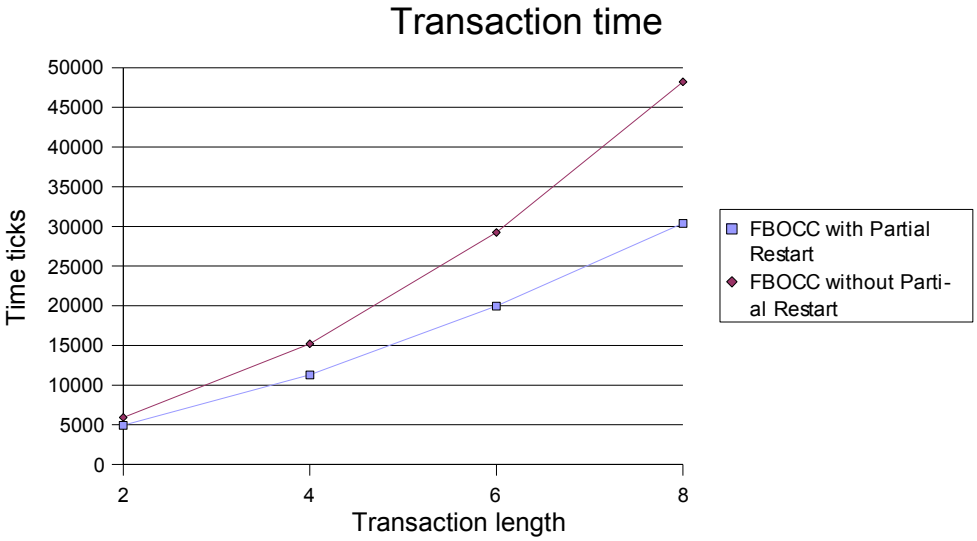


Illustration 6.3: Graph of FBOCC with and without partial restart

6.5.1.2 Client read-only with server transactions

We tested the read-only protocols with the time between each server transaction (inter server transaction time) set to 1000 and 5000 time ticks (Illustration 6.4).

PVTO and FBOCC had a too bad confidence interval for the longest transactions, so the values are not shown in the graphs. For inter server transaction time equal 1000 BCC-TI seems to give the best performance, while FBOCCpr gives the better performance for inter server transaction time equal 5000.

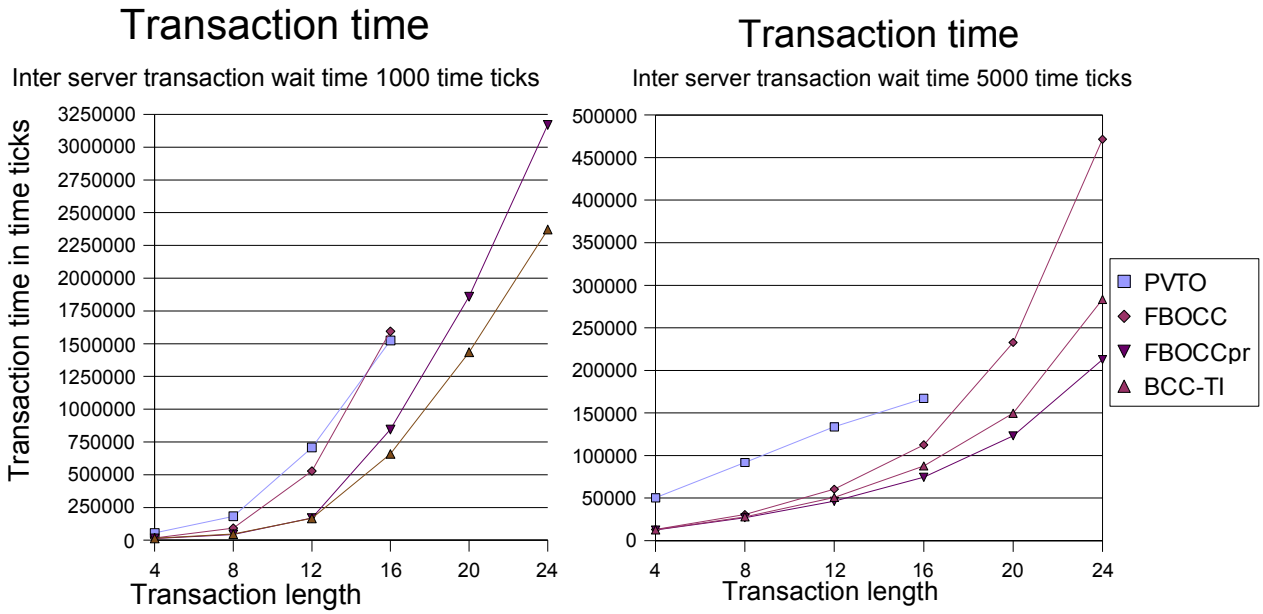


Illustration 6.4: Transaction time with inter server transaction set to 1000 and 5000 time ticks

For transaction lengths below 12, it seems like the performance is similar for all except PVTO.

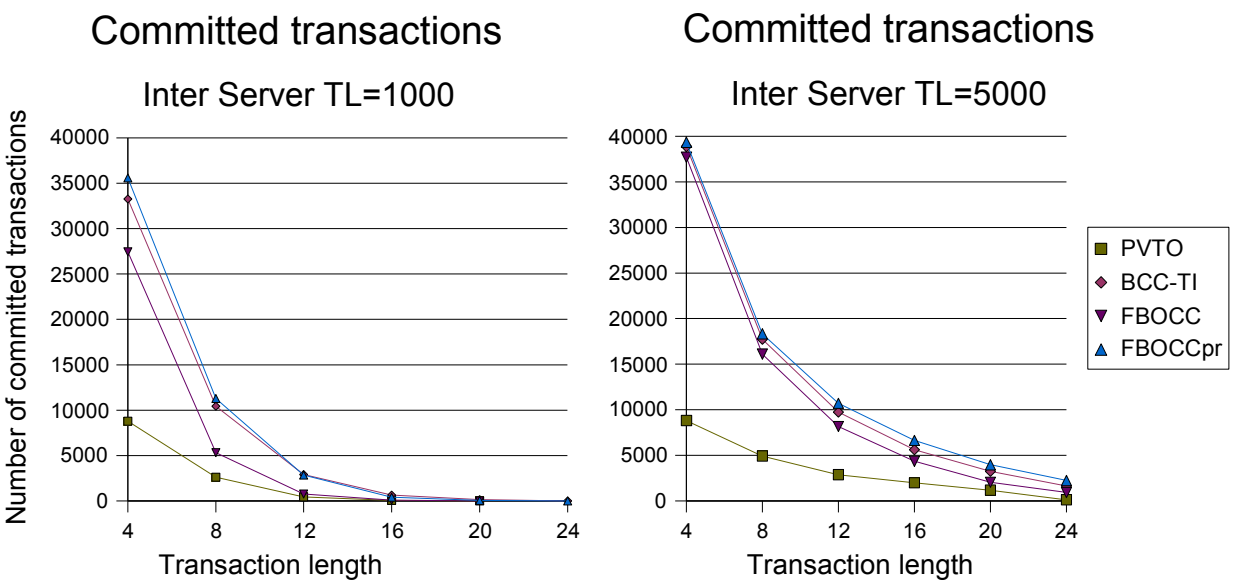


Illustration 6.5: Number of committed transactions for server transaction set to 1000 and 5000 time ticks

Illustration 6.5 shows the number of committed transactions during one simulation run. The results are very similar for all protocols except PVTO. The only thing worth to notice is that FBOCC struggles a bit more when the server transactions comes as often as each 1000 time tick. In general, the performance sinks drastically when the transaction length is increased.

6.5.1.3 Client Update Transactions Without Server Transactions

Illustration 6.6 shows how the transaction time increases when the transaction length increases. A linear increase in the transaction time is expected if there were no updates in the database set. To read 16 items would take twice the time it takes to read 8 items. But since the updates cause transaction aborts, and the likelihood for inconsistent data increases the longer a transaction lasts, the number of aborts increase. When the aborts increase, so do the transaction time before the transaction is committed. (Remember that all transactions are processed until they are committed, and when they get an abort then they restart.)

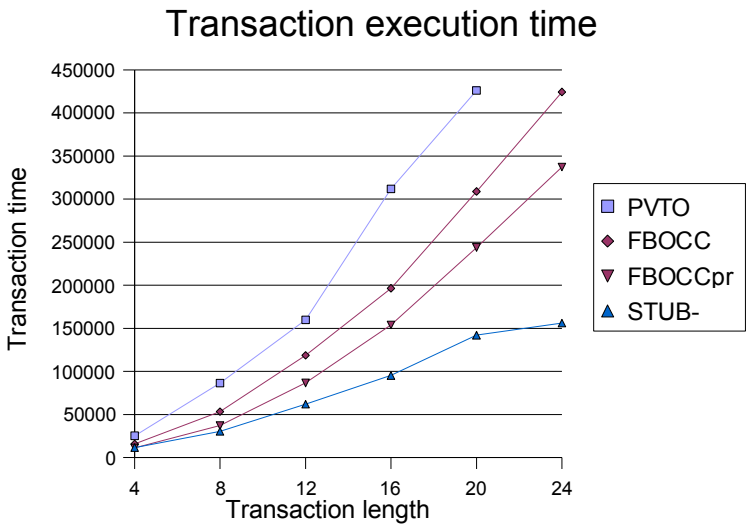


Illustration 6.6: Transaction time in standard environment without server transactions

The graph in Illustration 6.6 shows STUBcast as the one with lowest average transaction time. It increases almost linear, whereas the other increase quite rapidly when the transaction length exceeds 12. FBOCC with partial restart (FBOCCpr) decreases less then FBOCC. The reason why PVTO starts to flatten out

after transaction length 16 and STUBcast at length 24 is because of the bad confidence interval. In general, the values with transaction length over 16 should not be taken too seriously.

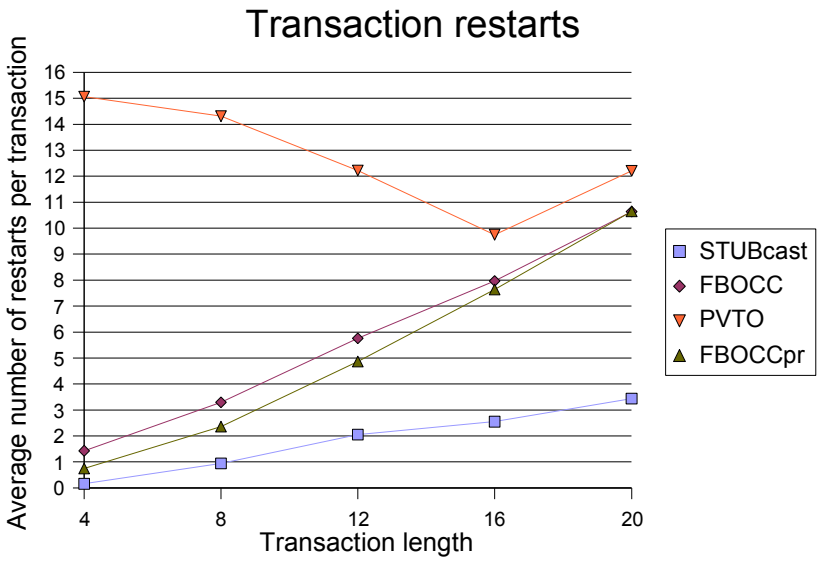


Illustration 6.7: Average number of restarts per transaction

Illustration 6.7 shows the average number of restarts per transaction. FBOCC has most restarts, and in average more then one restart per transaction already for transaction length 4. The rate for FBOCCpr is of course the average number of *partial* restarts per transaction. Each restart leads to longer transaction time, so STUBcast's efficiency can clearly be seen here. PVTO has extremely many restarts, even for low values of the transactions length. This might suggest some error in the implementation we have done, because on the paper it should perform better. On the other hand, in the performance results of PVTO we did not manage to understand how many clients were simulated. The protocol might perform good in environments with few clients.

The three graphs in Illustration 6.8 shows the amount of restarts done at the client and server side. The more restarts performed at the client, the better, because early abort reduce the waste of bandwidth, server processing and time. FBOCC have more than 80% restarts on client side, which indicates the use of a stricter validation criterion on the client side. The consequence is that a serializable transaction might be

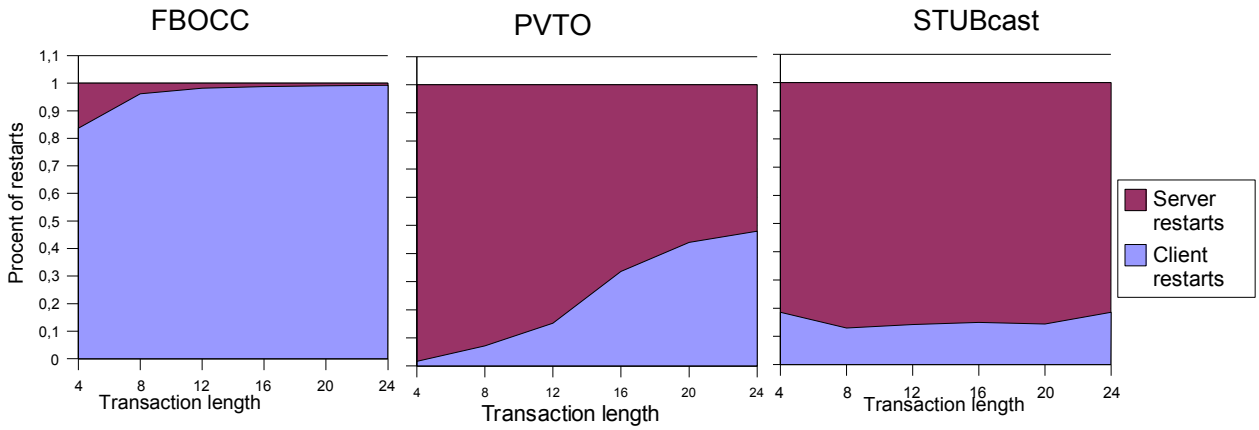


Illustration 6.8: Relationship between the amount of client restarts and server restarts

aborted, but when a transaction first gets through the validation at the client side, it has a very high probability of being accepted at the server.

STUBcast and PVTO aborts most transaction on the server which waste time etc.. In return, they have much less restarts overall.

The number of accepted transactions per simulation run measures the effectiveness and the saturation limit. That is, the amount of transactions that can be accepted. Illustration 6.9 shows FBOCCpr with most transactions committed and FBOCC and STUBcast following on a shared second place. PVTO has a very poor performance for all transaction lengths.

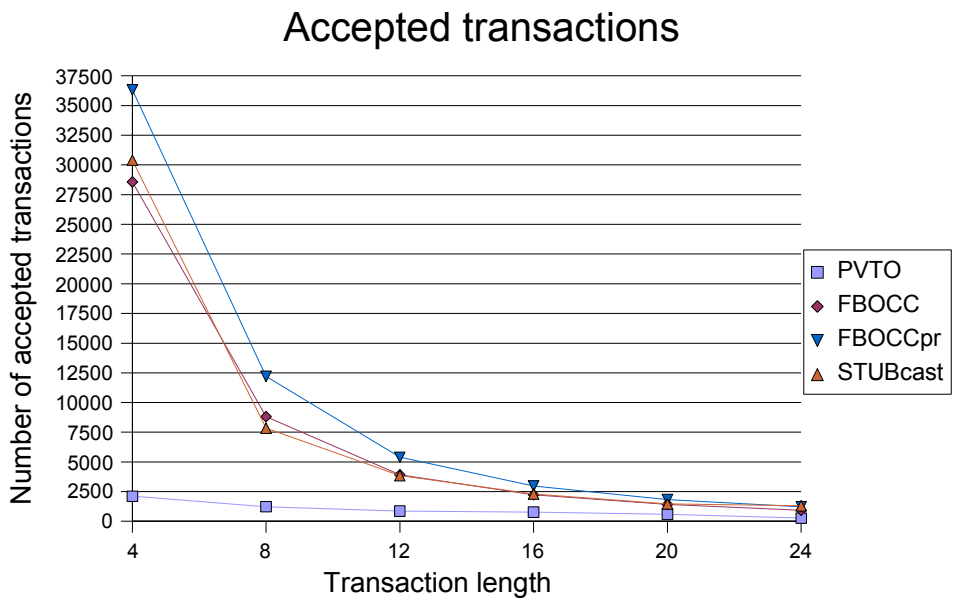


Illustration 6.9: Number of accepted (committed) transactions during one simulation run

It seems contradictory that STUBcast has the shortest average transaction time, while FBOCC has most transactions accepted during the simulation time. When the average time to get a transaction accepted is short, we assumed it would have most transactions accepted during the simulation time, because once one transaction is finished, the next can continue and so on. The explanation to the behaviour we see lays in the distribution of the transaction time, and is not seen in the average.

Illustration 6.10 shows the histogram of the transaction time of FBOCC, FBOCCpr and STUBcast with transaction length 6. The peak right before 20 000 time ticks is the same for all the protocols. Then the graph flattens out, and at the end both FBOCC and FBOCCpr raises quickly. That is only to illustrate the number of transactions that use *more than* 72 000 time ticks to complete a transaction.

STUBcast raises slower to the first peak right before 20 000 time ticks, and is also a bit slower to fall down. That means it has generally longer transaction time than the other. But at 30 000 time ticks it falls down and before 40 000 time ticks, STUBcast has 97,5% of its transactions. FBOCC on the other hand has only 80% of its transactions before 40 000 time ticks, and FBOCCpr has 89%. With other words FBOCC and

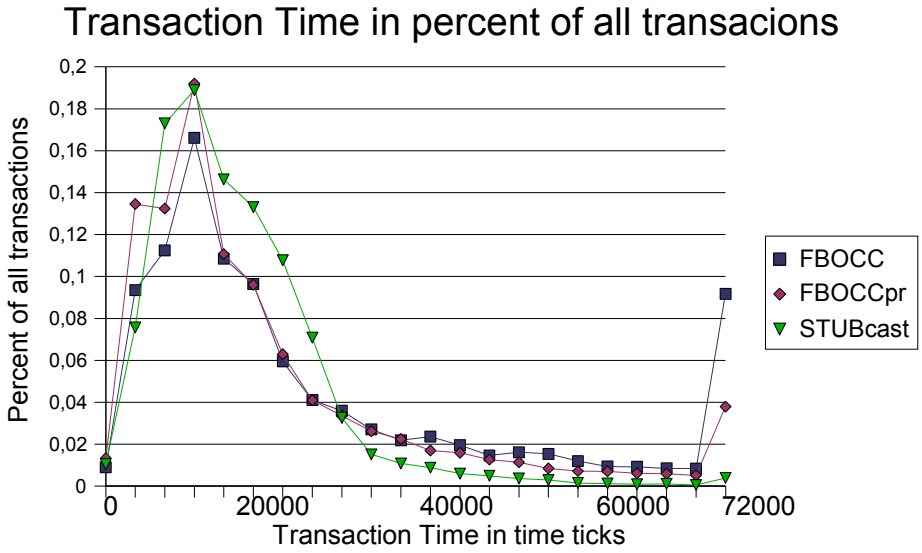


Illustration 6.10: Transaction time in percent of all transactions (Transaction time histogram)

FBOCCpr are very quick to commit a major part of its transactions (around 80%), but the rest are very long and contribute to a higher average. STUBcast is a bit slower (but still quick) than FBOCC on the quickest transactions and have very few transaction that takes a long time.

The effect is that FBOCC gets many transactions committed because the major part commits fast, but a minor part is very long and case a higher average. The fact that some transactions gets a very long transaction time, which is caused by many restarts, is disturbing. The phenomenon of constantly being restarted is not acceptable in most environments. It does not matter if 95% of the users experience an excellent service, as long as 5% must wait for ages to get a response.

6.5.2 Transaction Length for Qualitative Characterisation Framework

In order to have the same performance measurement criterion for all protocols, we tried to find an attribute to compare them all.

The problem of comparing them is that STUBcast do not have server transactions in our implementation, while BCC-TI needs server transactions in order to be measured (unless the database will be static and all transactions will be accepted). The trick is to find the right ratio for client update transactions, such that the number of updates are approximately the same as the server transactions in a read-only environment.

We tried FBOCC with standard environment settings but without server transactions, and we got results similar to FBOCC with server transactions in a read-only environment. (See Illustration 6.11 and Table 6.3). The values are far from an exact match, but they works as an indication of the transaction time.

BCC-TI is measured with inter server transaction time equal 5000.

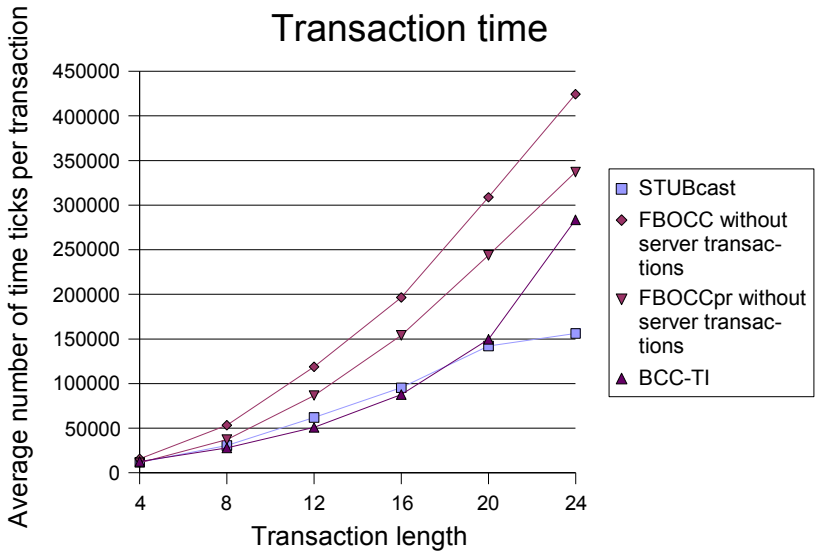


Illustration 6.11: Transaction time when the transaction length change

We use transaction time for transaction length 6 as the standard measurement.

Protocol	***With server transactions (RO)***	***Without server transactions***
STUBcast	NA	18 669
FBOCC	21 018	31 488
FBOCCpr	19 398	21 838
PVTO	67 503	57 300
BCC-TI	19 687	NA

Table 6.3: Mapping table for average transaction time with transaction length 6

Table 6.3 shows the error of our mapping from “read-only” environment to “no server transaction” environment. FBOCC have 10 000 time ticks less in the read-only environment where as PVTO has 10 000 time ticks more in the read-only environment. We are therefore satisfied with the mapping and believe we must tolerate the big error because of the protocol differences. In general, the mapping is difficult because the rate of the server transactions are quite stable, where as the rate of the client update transactions are very variable. The client update transaction are much fewer when many conflicts and restarts occur. PVTO has in general a very bad commit rate, and does therefore perform better without server transactions. FBOCC has a very good commit rate, and do therefore suffer without the server transactions.

The best solution would have been to implement server transactions in STUBcast, but on the other side, as with STUBcast, the authors did not specify how it should be implemented.

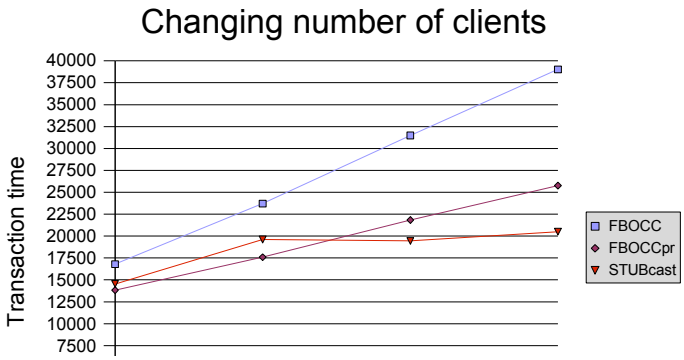
When values for both environments are available, we choose the values from without server transactions for the framework. We think the server transactions is simply a special case of a client, and feel it is a disturbing factor in our measurements (another aspect of uncertainty).

We want to emphasise the criterion's weaknesses as a performance measurement. A protocol's average transaction time for transactions with length 6 gives a poor description of the protocols performance. The criterion do not say anything about factors such as the distribution of the transaction time (continuous restart might occur often or not at all no matter what the average transaction time is), or maximum transactions before saturation. We only suggest this criterion in lack of better alternatives.

6.5.3 Number of clients

When the number of clients increase, the performance of the different protocols vary. We made a test to gauge how the performance is affected when this happens. Compared to the other protocols we implemented PVTO perform quite poor in our standard environment so it is omitted in this test. We also omit BCC-TI since it is read only and is therefore incomparable to the others. We therefore test FBOCC, FBOCCpr and STUBcast against eachother.

Illustration 6.12 Shows how the transaction time increase when the number of clients increases, it is important to note how the increment of the transaction time in STUBcast slows down compared to the others. This is because of the loose restriction on the serializability, while the other protocols suffer when the number of updates increases. We see that FBOCCpr outperform the normal FBOCC as the number of clients increase and with a low number of clients it even outperforms STUBcast.



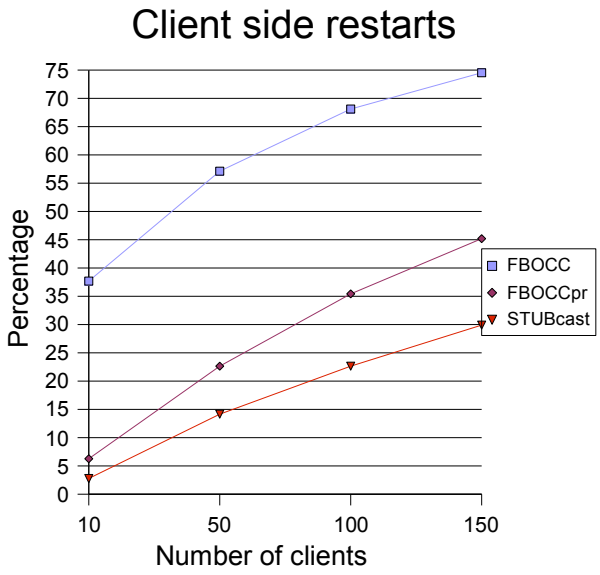


Illustration 6.13: Restarts when the number of clients increase

When we look at the restarts in Illustration 6.13 we can see that STUBcast fast reach a high percentage of restarts when the number of clients increases. This happens since there is such a loose restriction on the serializability as can be seen in the table over the client side restarts where STUBcast has by far the lowest amount of restarts. It is also worth noticing that in FBOCCpr we also have this trade-off where an decreased amount of client restarts result in more restarts on the server side.

Again we can conclude that the decreased serializability of STUBcast results in an increased performance in form of a reduced transaction time. We also see that FBOCCpr keeps an increasingly better performance than normal FBOCC as the number of clients increase.

6.5.4 Database size

With the increased database size the number of conflicts will decrease, but the item wait time will increase and therefore the transaction time will also increase. We did a test in our heavily updated standard environment to see how a change in the broadcast database size would affect the transaction time and the number of restarts.

From Illustration 6.14 we can see that there are big differences, especially when we have a small database. When the database size is 300 PVTO have reached the limit where the advantage of getting the data items faster is outweighed by increased number of restarts. It is also worth noticing that both FBOCC and FBOCC with partial restarts (FBOCCpr) have roughly the same transaction time when the database is small and we have a lot of restarts. But when the database size increases and the number of restarts goes down FBOCCpr gets near to the performance achieved by STUBcast

In Illustration 6.15 we can clearly see how a small database greatly increases the number of restarts, both on the client side and on the server side.

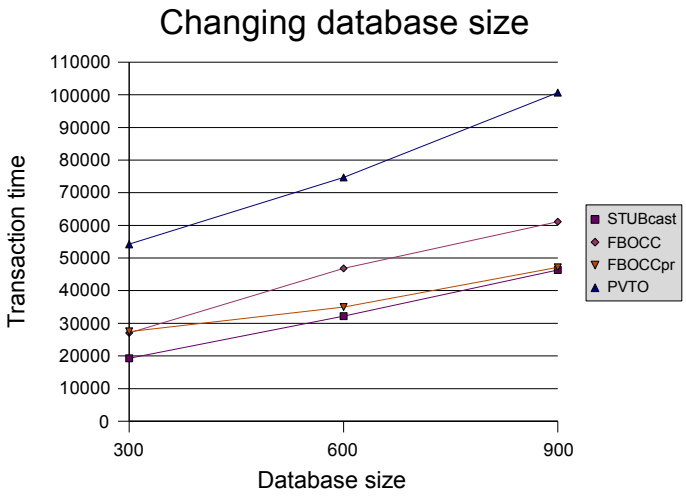


Illustration 6.14: Transaction time when the database size

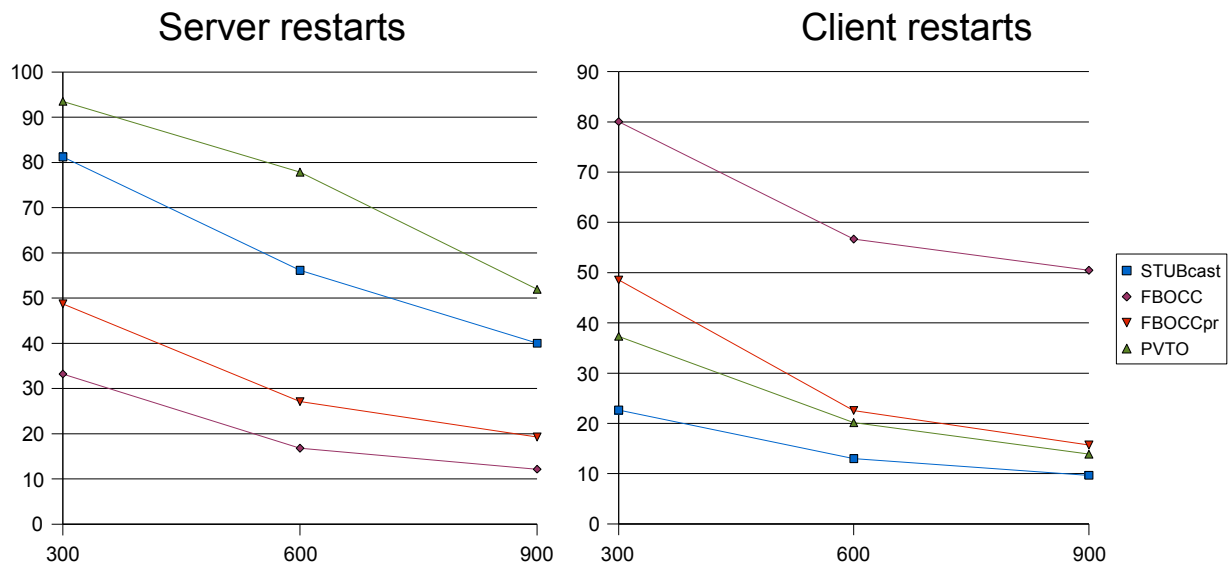


Illustration 6.15: Restarts when the database size increases

6.5.5 None uniform access distribution

Most real environments do not have a totally random access pattern so we implemented broadcast disks with some of the concurrency protocols. This to see wither it would affect the performance of some methods better than others. We look on BCC-TI and FBOCC, both with the standard environment mentioned earlier. But to be able to compare the results we used a read only environment, since BCC-TI is a read only protocol.

Illustration 6.17 shows the comparison between the average transaction time and it is seen that BCC-TI perform better than FBOCC both with and without broadcast disks and the performance gain from the usage of broadcast disks is almost the same in both protocols.

In Illustration 6.16 we see that the read only optimized BCC-TI keeps the average number of restarts per. transaction at a lower level than FBOCC. This explains why BCC-TI perform better, as we noticed when comparing the average transaction time.

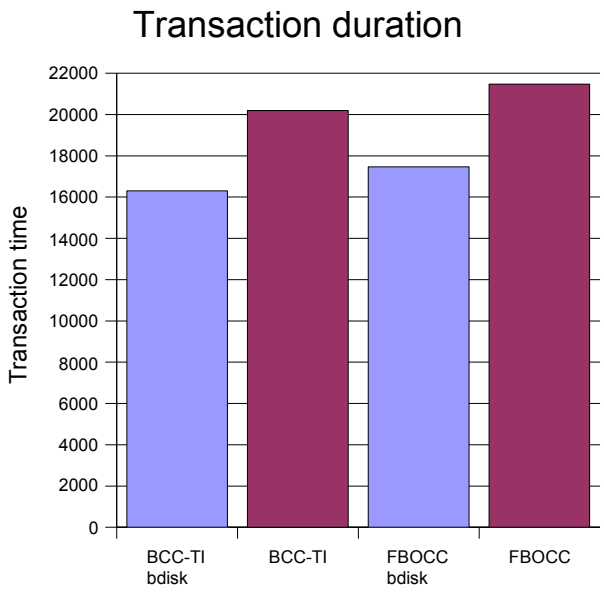


Illustration 6.17: Average transaction duration in a none uniform environment

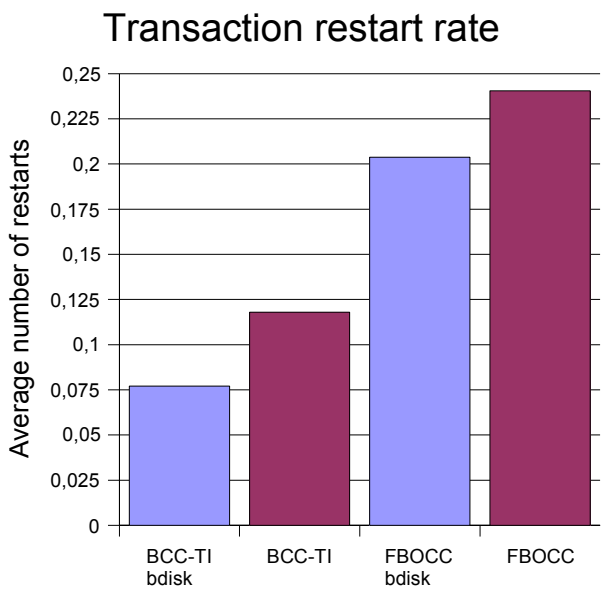


Illustration 6.16: Average transaction restart rate in a none uniform environment

We can conclude that broadcast disks can lead to a general performance increase by cutting down on the item wait time, but this might in some cases result in a longer broadcast cycle so that the control information used in some protocols gets delayed. It also raises the question if the control information should be sent after a major or a minor broadcast cycle.

6.6 Conclusion

First of all we can conclude that the drop in serializability used in STUBcast can decrease the latency. By doing so there will be a lot less conflicts and therefore more resistant to restarts in environments with a lot of updates and long transactions. In environments with a lot of restarts STUBcast manage to keep the number of server side restarts quite low, and do most of the restarts on the client side, saving the server from becoming swamped.

BCC-TI seems to perform very well when the environment is read-only and outperform the others in such environments, but STUBcast also performs almost equally well. When the drop in serializability is not an option FBOCCpr will perform the best. And the performance increment of FBOCCpr compared to normal FBOCC seems to increase with increasing transaction length.

7 Qualitative Characterization Framework

Based on the knowledge from the survey and simulation tests, we present a qualitative characterization framework. Its goal is to classify the various techniques to make it is easier to find the most efficient protocols for a specific environment.

The framework is divided into two distinctive parts, one describes the data management with the core functions like broadcast type, indexing and cache. The other describes the more complete concurrency control solutions.

7.1 Data Management

We define four different ways to categorize data management methods:

- Latency defines the total waiting time from an item is requested until it is fully received. A low value is better here as it reduces the total waiting time for a client. It is not possible to put a number on the latency unless you know the environment variables, but a general idea can be made to define whether it is high or low.
- Processing is how much CPU processing that is needed to perform the necessary calculations in the protocol. Little processing means less power usage at the clients and is therefore an important factor in battery powered devices, and devices with limited processing power.
- Size increase is how much the broadcast size will increase because of the method. Typically for indexing methods this varies a lot.
- Tuning time is how long time the client have to listen to the broadcast to be able to locate the requested item. The usage of network devices and processing of data has a high cost on some devices, it is therefore important to keep the tuning time as low as possible. This enables the client to sleep while waiting for the item to appear on the broadcast. This value will also vary a lot with the environments and is closely connected to the latency.

Table 7.1, 7.2 and 7.3 Shows the different characteristics of different data management parts. This again is divided into three parts, broadcast methods, indexing and cache. The following sections discuss the findings and gives a short explanation of its meaning.

7.1.1 Broadcast Methods

Out of the four methods three of them is made for the same environment. Broadcast disks and binary based broadcast is both made for environments where there is a none uniform access pattern where some data items are hotter than others. Broadcast disks is a bit easier to implement, and as long as the frequency of each data is chosen correctly it will perform equally good as the binary based method. But it is also the case that the binary based method will always perform equally good or better than broadcast disks, at the cost of a bit more complicated implementation. When it comes to the amount of processing needed for the methods they both perform equally well. At the price of more computation TC-AHB might be used but since it is a hybrid method it will need feedback from the client to perform good. It also performs good in environments where the access distribution may change over time. Something that both broadcast disks and binary based broadcast fail to adjust to.

In environments where a high rate of concurrency is needed multiversion might be used, if there are enough versions this method will have the highest grade of concurrency. But this is at the cost of an increased broadcast size and increased latency.

Method	*Latency*	*Broadcast size increase*	*Processing*
Broadcast disks	Decreases in none uniform distributions	Only by broadcasting hot data more frequently	Low
BNB	Decreases in none uniform distributions	Only by broadcasting hot data more frequently	Low
Multiversion	Some increase, but dependent of transaction length	Very large, but dependent of number of versions	Moderate
TC-AHB	Low for hot items	Fixed, but how much that is broadcasted on demand varies	Moderate

Table 7.1: Comparison of broadcast methods

7.1.2 Indexing Methods

Latency optimal is as the names says the indexing method that gives the best latency, simply by not having any index at all, but it is still a possible choice in environments where the tuning time is totally irrelevant.

The most useful methods are (1, m) and distributed indexing. (1, m) is very simple and easy to implement and gives a good performance. Distributed indexing is more complicated to implement, but offers a smaller increase in the broadcast size since it cuts down on the replication of data.

Method	*Latency*	*Tuning time*	*Broadcast size increase*	*Processing*
Latency optimal	Very low	Very high, same value as the latency	No size increase	None
Tuning optimal	Very high	Very low	small	Low
(1, m)	Average, but dependant on the value of m	Very Low	From small to big, dependent of the value of m. If optimal m is used then it is low	Low
Distributed Indexing	Average to low	Low	Depending on implementation, but generally of moderate size	Low

Table 7.2: Comparison of indexing methods

7.1.3 Caching Methods

Both PIX and PT is not easily implementable without making modifications to them because of the high processing needed. That leaves us with LIX, tag-team and LDF.

LIX performs very good in environments where there is a none uniform broadcast of data and the need for processing is low.

In environments where it is hard to predict the access pattern tag-team cache will help to greatly reduce the cost of a cache miss and therefore help a great deal on the latency.

And LDF performs very good when there exists deadlines that have to be meet.

Method	*Latency*	*Tuning time*	*Processing*
PIX	Decreases, especially good when used with broadcast disks	Decreases	Extremely high
LIX	Decreases, especially good when used with broadcast disks	Decreases	Low
PT	Decreases a lot	Decreases	Extremely high
Tag-team	Decrease, especially good when cache miss appears	Decreases	Low
LDF	Decreases, good when deadlines apply	Decreases	Moderately high

Table 7.3: Comparison of caching methods

7.2 Concurrency Control

To categorize concurrency control methods, we use the three characterisation terms defined in 5.3 Concurrency Control Protocols Characteristics and in addition we define the following:

- Electrical power usage is measured by a combination of amount of access to the broadcast channel, upload link and CPU processing. It is mainly decided by the amount of access to the broadcast channel, because all protocols are very conservative in the use of the upload link and they are pretty similar in the use of CPU.
- Performance should be measured with respect to latency, restart rate and total time to complete a transaction. These measurements varies dependent on the environment settings, so we will use the criteria and values defined in 6.5.2 Transaction Length for Qualitative Characterisation

Framework. The values are denoted Tt, for “Transaction time”. For protocols we have not tested, we will use a text based on the performance presented in scientific papers.

Some of the reviewed protocols we know little about the performance criteria because we did not do any simulations measurements, and no other results were found. Those protocols are therefore not included in the Table 7.4.

Method	*Electrical power usage*	*Performance*	*Real-time*	*Client update transaction*	*Correctness criteria*
F-matrix	Big CI leads to more accesses on bc-channel	Poor	No	Yes	Mutual consistency and currency
BCC-TI	Conservative	Very good, Tt=19867	No	No	Serializability
STUBcast	Clients must listen to the broadcast channel constantly during an active transaction	Very good Tt=18669	No	Yes	Single and local serializability
PVTO	Client must use uplink for read-only transactions.	Poor, Tt=57300	Yes	Yes	Serializability
FBOCC	Conservative	OK, Tt=31488	Yes	Yes	Serializability
EOCC	Conservative	Better than FBOCC (OK)	Yes	Yes	Serializability

Table 7.4: A framework for concurrency control protocols

7.2.1 Electrical Power Consumption

Constant listening to the broadcast channel during an active transaction is required by some of the protocols because of the non static size of the broadcast, and the fact that update messages can be broadcasted at any time. The new updated data is faster disseminated in such cases, but the power consumption at the client side is increased. Thus, those approaches (UFO and STUBcast) are not suitable for battery powered handheld devices. Another backside is the vulnerability for bad data channel. If the client loose the signals for a small moment during an active transaction, the client do not know if an update has been broadcasted or not. In the case of updates being broadcasted in CI at scheduled time, it is only important to listen at this exact point of time. Of course, signal loss at this time will cause the same problems, but the probability for signal loss during one time unit is smaller then for loss during one hundred time units.

In Table 7.4, *conservative* means the amount of accesses to the broadcast channel and upload link is at a minimum and so is the demand for processing power. Conservative is the best value, which means one of the least power consuming. As previously stated, the CPU processing has little impact on the power consumption because the CPU usage is pretty similar for all the protocols, and the same goes for the use of the upload bandwidth.

7.2.2 Validation Algorithm Complexity

The complexity of the validation algorithm do not count for the clients, but it do counts for the server. If the server receives many validating transactions, it will be overloaded faster if the validating algorithms are complex. Also the response time will increase. Some protocols are quite complex, but with indexing and other optimization techniques it is difficult to know how complex without real implementation. We decided not to dwell upon this topic, but some work should be done to see how many transactions a server can handle before congestion.

8 Discussion

This chapter summarize the report, and look at the work we have done. The work is compared up against our initial task description in 8.1, and in 8.2 the work is evaluated for relevance and reliability.

8.1 Project Outcomes

This project have several contributions to the research field, namely the survey, characterization framework, simulation platform with many implemented techniques, and the proposal of a new technique that optimizes the transaction restart.

8.1.1 Survey

The first requirement in our task description was to survey the topics of data management and concurrency control in broadcast based asymmetric environments. We have done that by focusing on techniques used for performance increase. The survey is relevant for people totally new to the area because it gives a soft introduction. Nevertheless some parts are more technical, and is directed on people with some experience within the field. The reader new to the topic can skip these parts. The techniques are surveyed and summarized.

8.1.2 Qualitative Characterization Framework

As part of the survey, some criteria describing the characteristics to the protocols were found. These criteria are useful for categorizing and were used in the characterization framework. The categories makes it easier to compare techniques against each other and select techniques for a specific environment. It is also easy to see which protocols are the best fit for each environment, just by looking at the framework. But since we did not simulate all solutions we do not know the performance of all the protocols. The framework was made on the knowledge from surveying the topic and the results from the simulations we performed, which was a requirement from the thesis description.

8.1.3 Simulations

Our thesis description stated we would make a simulation platform in CSIM, and that we did. The simulation platform is a tool that can be used in further research within the area. With the basic framework, new protocols can be implemented and compared with the already implemented protocols.

The simulation tests we performed gave us deeper knowledge of the protocols. We chose to implement a wide diversity of protocols. Within concurrency control we chose one read-only protocol (BCC-TI), one with reduced correctness criteria (STUBcast), and two quite new protocols that are for all environments (FBOCC and PVTO). In addition we implemented an improved restart technique in FBOCC which we named *partial restart*.

In read-only environment BCC-TI had the shortest average transaction time. FBOCC with partial restart followed right behind, where as normal FBOCC was slightly worse. PVTO did bad in all our tests. STUBcast had the shortest average transaction time in environments without need for serializability, real-time transactions, battery powered devices, and server transactions (because STUBcast did not implement that in our simulations). But FBOCC with partial restart has actually more accepted transactions during one simulation run, and normal FBOCC has the same as STUBcast. Still, we think STUBcast performs the best, because it is most important with short transaction time. The distribution of the transaction times was also more concentrated than FBOCC. For all other environments FBOCC is the best, and of course FBOCC with partial restart is even better.

We did not implement EOCC, but we know from Li et al. [53] it performs better than FBOCC. It is hard to say if it performs better that STUBcast, but it is a chance it does. EOCC and FBOCC is a protocol that supports the strongest criteria, such as restrictive battery usage, serializability, client update transactions, and real-time. This makes it applicable to all environments in question. If EOCC performs better than BCC-TI and STUBcast, it is most likely the best performing concurrency control protocol for all the environments. That makes it easier to choose protocol.

Our new proposed technique, partial restart, shows great performance increase on the transaction time. To implement *partial restart*, the protocol must be able to know which operation caused the conflict. BCC-TI, STUBcast and PVTO have some problems knowing this since they use a timestamp interval to indicate a conflict. Some modifications must be done in the implementation in order to use the *partial restart*.

8.2 Evaluation

This survey introduce and review the work done the last decade. The previous survey touching this topic was published 7 years ago, so we think this survey is a useful contribution. The focus on both data management and concurrency control leads to a good integration of these two sub-areas. Little work that combines these techniques have been found, although the topics are closely related.

The categorizing done in the characterization framework is useful because it gives a better overview. It presents the protocols in such a way that they are easy comparable. The criteria we chose for categorizing are based on the limiting factors in the protocols. We do not claim we have chosen the most correct criteria, so they should be viewed as a suggestion open for improvement. At least the measurements can be improved, especially for the *performance* criteria under concurrency control.

We had visions of constructing a unique way to measure the performance, but since the protocol properties are so changing, it turned out to be difficult. For example BCC-TI and STUBcast is not directly comparable, because BCC-TI relies on server transactions to get conflicts (remember that clients are read-only in BCC-TI) and STUBcast do not have server transactions. One solution is to implement some server transactions in STUBcast too. Only one performance measuring unit is most likely to little descriptive because there are so many changing variables.

We finally ended up with *average transaction time* for a standard environment (transaction length = 6, database size = 300, server transaction delay = 5000, etc.). Read-only protocols were measured with server transactions and update protocols were measured without server transactions. We think it is describing enough, although it does not tell anything about the maximum transaction time some clients may experience.

The simulation platform we made worked fine for our protocol implementations. The protocols varies quite much, so many changes is necessary from one protocol to another, but a framework remains. Our framework contains the processes, mailboxes and structures (showed in Illustration 6.7). In addition it contains the statistics measurement (which must be placed in the right place in the code to measure right values), the cyclic broadcast and the transaction creation.

In all of the simulations we use CSIM's built-in ability to generate a confidence interval. This to be confident that our data is within a acceptable range. In some protocols it were hard to acquire a good confidence interval with long transaction lengths. In some cases it just took to long to acquire good enough results to get a good confidence interval. One reason was because the protocol had so many rejections that very few transactions were committed during a long time interval. Then the confidence interval concerning number of transactions committed were very bad. Other reasons were the big variance in some sample data which would require a very large number of data to be collected.

We performed test with the protocols with some extreme values on the environment variables. The tests was on transaction length, number of clients, database size and non-uniform client access distribution.

The transaction length went from 4 to 24 with steps of 4 operations. With 100 clients and time between each server transaction at 1000, the environment was very demanding. We wanted to see a congestion, and we did. Only the scientific paper presenting STUBcast had shown simulation results with such high transaction length. Most applications do not need such many operations in one transaction, so the relevancy can be argued. But by testing extreme, more knowledge of the behaviour is achieved. After the tests were done, we did realize that the longest transactions were unnecessary to test for, because the confidence interval was very poor. Full congestion or very near full congestion occurred. Still, it was useful to see this.

We think the tests on the number of clients are very relevant, because no other simulations report any number of clients or similar. We are probably the first to make a simulation platform simulating multiple clients. Most of the others use one multi threading client that runs many transactions concurrently. The end effect can be the same, if the client runs each transaction thread independently the whole simulation run. If the transaction threads are started on new and ended for each transaction, then the natural distribution of the transactions, with bursts and then no traffic, is difficult to simulate. Nevertheless, the number of transaction threads are not given, only the time between each transaction.

Our tests was with 10, 50, 100 and 150 clients. These numbers do not represent a normal number of clients, because our clients produce very many transactions in order to stress the environment. A normal number would have been 1000 or 10 000 dependent on the access pattern for each client. But such many clients would have taken very long time to simulate. We therefore decided to have more active and fewer clients. The changing in number of clients gives a good impression of the scalability with regards to the number of clients.

The simulation tests use a standard environment with a database consisting of 300 data items. That is a very small database, but helpful to create a hotspot effect and a stressful environment without increasing the simulation execution time. Just to show other values, we tested for size 300, 600 and 900 database items. The transaction time increase steadily with bigger database, because the broadcast cycle gets bigger and needs more time to do a cycle.

The performance increase with *partial restart* in FBOCC is especially high in our tests because the environment have a very high load with many restarts naturally. The more restarts, the bigger will the improvement be with partial restart. In an optimal environment, there will be no restarts and in such a case *partial restart* will have no performance effect at all. But an optimal environment will never occur in practical cases, and even tough most transactions are not restarted, some transactions are unlucky and might experience several restarts. Even in environments with little load, some few transactions had several consecutive restarts. Therefore we argue that it is always useful to use partial restart whenever possible. The performance will not always increase so much, but it will always increase or in worst case be equal.

9 Conclusion

The survey gives an introduction to this research topic, and focus on the recent advances within techniques for performance increase. The focus on the techniques lead to the development of *partial restart*. which leads to almost 30% faster transaction execution time when the clients have a transaction length of 6 operations. The longer the transaction length is, the better performance increase is given by the *partial restart*.

Protocols are categorized such that it is easier to see which environments or qualities they adhere to. This work showed us that the most general protocols, that is, those who apply to most environments, do actually perform very good. In read-only environments BCC-TI performs best, but FBOCC with partial restarts performs almost equally well. Under normal update environments FBOCC (and FBOCCpr) is the protocol that performs best, but if a relaxed serializability criteria is possible STUBcast manage to perform even better. But STUBcast have the disadvantage of demanding that the client must listen to the broadcast channel continuously during an active transaction. Therefore it is not suitable for environments with battery powered handheld devices.

This report presents some performance results which gives an impression of the effectiveness of the various protocols. Makes it easier to choose and make data management and concurrency control protocols for a specific environment and application. Nevertheless we can not say exactly when to use one thing, and when to use another. It aims at giving the reader an understanding and hopefully intuition for what to choose, depending on the type of application and environment. The reader can find more in-depth information in the references about the protocols presented.

9.1 Further Work

An interesting topic for further research would be to identify when the different protocols gets saturated. At some point there will be to many updates and the number of restarts will become intolerable. This would of course be dependent of the transaction length, database size and the number of updates. It is also possible that there exist some limit on how many transactions a server can handle, dependent of how much processing that is needed to validate the transactions on the server side.

When we implemented the protocols in CSIM we experienced many errors. In order find more errors and to guarantee the correctness of the protocol we would have liked to implement a check of the correctness criteria. The simulations provides test results that appear to be valid and CSIM indicate no errors during runtime. But due to the complexity of the models, errors might exist that affect the results in some way or another, even though great heed has been taken to prevent it. A correctness layer could help in insure the correctness of the simulation code, to avoid getting incorrect simulation results due to implementation errors. If a protocol guarantees serializability, then the global time tick should be associated with each operation and checked for serializability. Then it is also possible to measure if a protocol rejects transactions which in reality are serializable.

In most environments it is more applicable to use a hybrid version with both push and pull based data dissemination than a pure pull based. It would be interesting to do some research on how this could be implemented in combination with different concurrency protocols and how it would affect the performance.

We would also have liked to implement more protocols, especially EOCC so we could have done some more comparisons. We could also have looked for improvements by combining various techniques.

In addition we would have liked to do more extensive work on the simulation platform. The environment should be configurable to make it more realistic. For instance it should be easy configurable to change settings such as a delay on the upload link and the broadcast channel. Some clients close to the base station would get the signals earlier then some clients far away. Real life values should be used.

The optimistic concurrency control protocols suffer from many restarts. We did not write so much about it, but even though the load is low, one transaction might restart several times just because it is unlucky. In terms of probability, this can happen all the time no matter what the load is, only the probability might be a bit smaller. We present *partial restart* as one improvement for restarts, but other approaches should also be found in order to avoid a transaction being constantly restarted. Some prioritizing scheme could be made for transactions that already restarted once. The partial restart is a future research area because an general approach supported by a framework should be made to simplify the implementation in any protocols.

A protocol might look good on the drawing board or in the simulations, but in real life unforeseen complications usually arise. Best of all is a real implementation of the protocols. The survey by Barbara [11] also announced this need.

Finally, EOCC which is possibly the best protocol available for any environment also suffer from some fake conflicts. So even more improvements are possible.

Appendix

A1 Glossary & Abbreviations

Short definitions of terms used and references to further relevant glossary sources

Ad Hoc	Network constructed between peers without any managed infrastructure
ADSL	Asymmetric Digital Subscriber Line
Asymmetric	Bandwidth from server to client is bigger than from client to server
Backchannel	A channel back, either from server to client or client to server, dependent on the context
BCC	Broadcast Concurrency Control
BCC-TI	Broadcast Concurrency Control with Timestamp Interval
BCC-FW	Broadcast Concurrency Control with Forward Validation
Broadcast	One peer can reach all other local peers with one message
CC	Concurrency Control, used to keep the dataset consistent
CSIM	Commercial software for simulation
CI	Control Information, used in *CC* to inform clients of recently updated data items
Consistent data	Keep the data as if all transactions execute serially and not concurrently
DBMS	Data base management system
Dissemination	Distribution of data from one (or more) central source(s) to very many users
DM	Data Management, techniques for structuring the broadcasted data
EOCC	Efficient Optimistic Concurrency Protocol, a concurrency control protocol for broadcast env.
FBOCC	Forward Backward validation *OCC*, a concurrency control protocol for broadcast env.
History	A sequence of *transactions*
Index	Used in *DM* to inform the clients where to find a data item in order to decrease *tuning time*
Latency	Time it takes for a client to find and receive a data item
OCC	Optimistic Concurrency Control, non-lock based concurrency control protocol
OCC-BV	*OCC* Backward Validation, validates up against already committed transactions
OCC-FV	*OCC* Forward Validation, validates up against the currently active transactions
Operation	A read or write action on one data item
Partial Restart	Technique used to optimize the restart in OCC protocols, by only restarting the invalid part
PDA	Personal Digital Assistant, small handheld computer
PVTO	Partial Validation and Timestamp Ordering, another *BCC* protocol
Serializability	A correctness criteria requiring the execution of a *history* to be equal a serial execution
SGG	Stored Serialization Graph, contains committed transaction history and is used to validate new transactions
Transaction	*Operations* with dependences, an *operation* is decided from the value of the preceding one
Transact. time	Time it takes for a client to complete a transaction from start to successfully commit
Trans. length	Number of operations in a transaction
Tuning time	Time it takes for a client to find only the location of a data item
UFO	Update First with Ordering, a read-only *BCC* protocol
WLAN	Wireless Local Area Network

A1 References

1: A. J. Biggs. "Teletext systems: a review". Physics in Technology 10, Page 11-19, 1979

2: G. Childs. "United Kingdom Videotex Service and the European Unified Videotex Standard". British Telecom Res. Labs, Suffolk, England, Page 245-249, 1983

3: G. Herman, G. Gopal, K. C. Lee, A. Weinrib. "The Datacycle Architecture for Very High Throughput Database Systems". Proc. ACM SIGMOD Conf., San Francisco, Page 97-103, May 1987

4: T. Imielinski, S. Viswanathan, BR. Badrinath. "Energy efficient indexing on air". ACM SIGMOD Record, Page 25-36, 1994

5: S. Acharya, R. Alonso, M. Franklin, S. Zdonik. "Broadcast disks: data management for asymmetric communication environments". In Proceedings of ACM SIGMOD Conference, Page 199-210, 1995

6: Y. I. Chang, C. N. Yang, J. H. Shen. "Binary-Number-Based Approach to Data Broadcasting in Wireless Information". International Conference on Wireless Networks, Communications and Mobile Computing, Page 1, 2005

7: J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, K. Ramamritham. "Efficient Concurrency Control for Broadcast Environments". ACM SIGMOD International Conference on Management of Data, Page 85-96, June 1999

8: H. T. Kung, J. T. Robinson. "On optimistic methods for concurrency control". ACM TODS, Page 213-226, June 1981

9: L. Guohui, Y. Bing, C. Jixiong. "Efficient optimistic concurrency control for mobile real-time transactions". Proceedings. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and App, Page 443-446, August 2005

10: V. C. S. Lee, K. W. Lam, T. Kuo. "Efficient validation of mobile transactions in wireless environments". Journal of Systems and Software 69(1-2), Page 183-193, January 2004

11: D. Barbara. "Mobile Computing and Database--A Survey". IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, Page 108-117, January/February 1999

12: S. Acharya, M. Franklin, S. Zdonik. "Balancing push and pull for data broadcast". ACMSIGMOD '97, Page 183-194, 1997

13: G. H. Forman, J. Zahorjan. "The Challenges of Mobile Computing". IEEE Computer, 27(6), Page 38-47, April 1994

14: E. Pitoura, B. Bhargava. "Building Information Systems for Mobile Environments". Proceedings of 3rd International Conference on Information and Knowledge Management, Page 371-378, 1994

15: D. Linden, T. B. Reddy. "Handbook of Batteries (3rd Edition)". McGraw-Hill, 2003

16: Intel. "Electrical, Mechanical, and Thermal Specification". Intel® PXA270 Processor, Page 1-126, 2005

17: Philips Semiconductors. "Complete, single-package 802.11g solution for mobile phones & portable cons". BGW211 Low-power WLAN SiP, Page 1-4, 2005

18: K. Lam, M. Au, E. Chan. "Broadcast of Consistent Data to Read-Only Transactions from Mobile Clients". Proceedings of Second IEEE Workshop on Mobile Computing Systems and Applications, Page 193-204, 1999

19: P. A. Bernstein, V. Hadzilacos, N. Goodman. "Concurrency Control and Recovery in Database Systems". Addison-Wesley Publishing Company, 1987

20: T. Imielinski, S. Viswanathan, BR. Badrinath. "Data on air: organization and access". Knowledge and Data Engineering, IEEE Transactions on, Page 353-372, 1997

21: Philips Semiconductors. "Complete, single-package 802.11g solution for mobile phones & portable cons". BGW211 Low-power WLAN SiP, Page 1-4, 2005

22: J. Ebert, S. Aier, Gr Kofahl, A. Becker, B. Bur. "Measurement and simulation of the energy consumption of an WLAN interface". Technical University Berlin Telecommunication Networks Group, Page 1-23, 2002

23: A. G. Bar-Noy, B. G. Patt-Shamir, I. G. Ziper. "Broadcast Disks with Polynomial Cost Functions". Wireless Networks, 2004 - Springer, Page 157-168, 2004

24: M. H. Ammar, J. W. Wong. "On the Optimality of Cyclic Transmission in Teletext Systems". IEEE Transactions on computing, Page 68-73, 1987

25: O. Shigiltchoff, PK. Chrysanthis, E. Pitoura. "Adaptive multiversion data broadcast organizations". Information Systems, 2004 - cs.uoi.gr, Page 26, 2004

26: JJ. Fernandez, KJ. Ramamritham. "Adaptive Dissemination of Data in Time-Critical Asymmetric Communication En". Mobile Networks and Applications, 2004 - Springer, Page 491 - 505, 2004

27: E. Pitoura, PK. Chrysanthis. "Scalable processing of read-only transactions in broadcast push". Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on, Page 432-439, 1999

28: S. Acharya, M. Franklin, S. Zdonik. "Disseminating Updates on Broadcast Disks". Proceedings of the 22th International Conference on Very large databases, Page 12, 1996

29: S. K. Lee, M. Kitsuregawa, C. Hwang. "Efficient Processing of Wireless Read-only Transactions in Data Broadcast". Proceedings of the 12th International Workshop on Research Issues in Data Engineering, Page 101-111, February 2002

30: S. Zdonik, R. Alonso, N. Franklin, S. Acharya. "Are "disks in the air" just pie in the sky?". Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on, Page 12-19, 1994

31: S. Acharya, M. Franklin, S. Zdonik. "Dissemination-based data delivery using broadcast disks". ersonal Communications, IEEE, Page 50-60, 1995

32: Y. Huang, YH. Lee. "Caching Broadcasted Data for Soft Real-Time Transactions". Department of Computer Information Science Engineering University of Florida, Page 6,

33: C. Boksenbaum, M. Cart, J. Ferrié, J. F. Pons. "Concurrent certifications by intervals of timestamps in distributed databas". IEEE Transactions on Software Engineering, Page 409-419, 1987

34: E. Pitoura, PK. Chrysanthis. "Scalable processing of read-only transactions in broadcast push". Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on, Page 432-439, 1999

35: R. H. Thomas. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Data". ACM Transaction on Database Systems, 4 (2), Page 180 - 209, June 1979

36: T. Härder. "Observation on Optimistic Concurrency Control Schemes". Information Systems, 9(2), Page 111-120, June 1984

37: V. C. S. Lee, K-W. Lam, S. H. Son. "Real-time transaction processing with partial validation at mobile clients". Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA'00), Page 473-477, December 2000

38: H. Garcia-molina, G. Wiederhold. "Read-only transactions in a distributed database". ACM Transactions on Database Systems, 7(2), Page 209-234, June 1982

39: K. W. Lam, S. H. Son, V. Lee, S. L. Hung. "Using Separate Algorithms to Process Read-Only Transactions in Real-Time Sy". Technical Report 98-10, Department of Computer Science, City University of Hong Kong, Page 50-59, 1998

40: J. A. Stankovic, S. H. Son, J. Hansson. "Misconceptions about Real-Time Databases". IEEE Computer, 32(6), Page 29-36, June 1999

41: H. Guo, P-Å. Larson, R. Ramakrishnan, J. Goldstein. "Relaxed currency and consistency : How to say "good enough" in sql". SIGMOD Conference Paris, France, Page 815-826, June 2004

42: Y. Huang, Y. Lee. "STUBcast - Efficient Support for Concurrency Control in Broadcast-based Asy". Submitted to ICCCN 2001, Page 6, 2001

43: U. Lee, B. Hwang. "Optimistic Concurrency Control Based on Timestamp Interval for Broadcast En". Proceedings of the 6th East European Conference on Advances in Databases and Information Systems, Page 106-119, September 2002

44: V. C. S. Lee, K-W. Lam, S. H. Son. "Maintaining data consistency using timestamp ordering in real-time broadcas". Proceedings of the 6th IEEE International Conference in Realtime Computer Systems and Applications, Page 29-36, 1999

45: V. C. S. Lee, K-W. Lam, S. H. Son, E. Y. M. Chan. "On transaction processing with partial validation and timestamps ordering i". IEEE Transactions on Computers, 51(10), Page 1196-1211, 2002

46: T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, A. Weinrib. "The Datacycle Architecture". Communications of the ACM, vol. 35, no. 12, Page 71-81, 1992

47: P. M. Bober, M. J. Carey. "Multiversion Query Locking". Proceedings of the VLDB Conference, Vancouver, Canada, Page 1-34, August 1992

48: W. Weihl. "Distributed Version Management for Read-Only Actions". IEEE Transactions on Software Engineering, 13(l), Page , January 1987

49: V. C. S. Lee, K-W. Lam, S. H. Son. "Concurrency Control Using Timestamp Ordering in Broadcast Environments". The Computer Journal 45(4), Page 420-422, 2002

50: D. Barbara. "Certification Reports: Supporting Transactions in Wireless Systems". Proc. IEEE Int. Conf. Distributed Computing Systems, Page 466,

51: A. Das, K. Y. Kai. "Tradeoff between Client and Server Transaction Validation in Mobile Environment". Proc. IEEE Int. Database Engineering & Application Symposium, Page 265-272, 2001

52: E. Pitoura. "Supporting Read-Only Transactions in Wireless Broadcasting". Proceedings of the DEXA98 International Workshop on Mobility in Databases and Distributed Systems, Page 428-433, August 1998

53: G. Li, B. Yang, J. Chen. "Efficient optimistic concurrency control for mobile real-time transactions". Proceedings. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and App, Page 443-446, August 2005

54: Texas Instruments Incorporated. "Low Power Advantage of 802.11a/g vs. 802.11b". White Paper, Page 1-11, December 2003

55: D. Fussell, Z. M. Kedem, A. Silberschatz. "Deadlock Removal Using Partial Rollback in Database Systems". Proc. ACM SIGMOD int. conf. on Management of data, Page 65-73, 1981

56: G. Li, H. Wang. "A novel min-process checkpointing scheme for mobile computing systems". Journal of Systems Architecture: the EUROMICRO Journal, Page 45-61, 2005

57: H. Schwetman. "CSIM18 - The Simulation Engine". Simulation Conference Proceedings, 1996, Page 5, 1996

58: H. Schwetman. "CSIM User's Guide (Version 18)". MCC Corporation, Page , 1998

59: OpenOffice.org. "Open Office 2". , Page , 2006